Đinh Mạnh Tường

Giáo trình

Trí tuệ Nhân tạo

Khoa CNTT - Đại Học Quốc Gia Hà Nội

Phần I
Giải quyết vấn đề bằng tìm kiếm

Vấn đề tìm kiếm, một cách tổng quát, có thể hiểu là tìm một đối tượng thỏa mãn một số đòi hỏi nào đó, trong một tập hợp rộng lớn các đối tượng. Chúng ta có thể kể ra rất nhiều vấn đề mà việc giải quyết nó được quy về vấn đề tìm kiếm.

Các trò chơi, chẳng hạn cờ vua, cờ carô có thể xem như vấn đề tìm kiếm. Trong số rất nhiều nước đi được phép thực hiện, ta phải tìm ra các nước đi dẫn tới tình thế kết cuộc mà ta là người thắng.

Chứng minh định lý cũng có thể xem như vấn đề tìm kiếm. Cho một tập các tiên đề và các luật suy diễn, trong trường hợp này mục tiêu của ta là tìm ra một chứng minh (một dãy các luật suy diễn được áp dụng) để được đưa đến công thức mà ta cần chứng minh.

Trong các lĩnh vực nghiên cứu của **Trí Tuệ Nhân Tạo**, chúng ta thường xuyên phải đối đầu với vấn đề tìm kiếm. Đặc biệt trong lập kế hoạch và học máy, tìm kiếm đóng vai trò quan trọng.

Trong phần này chúng ta sẽ nghiên cứu các kỹ thuật tìm kiếm cơ bản được áp dụng để giải quyết các vấn đề và được áp dụng rộng rãi trong các lĩnh vực nghiên cứu khác của **Trí Tuệ Nhân Tạo**. Chúng ta lần lượt nghiên cứu các kỹ thuật sau:

- Các kỹ thuật tìm kiếm mù, trong đó chúng ta không có hiểu biết gì về các đối tượng để hướng dẫn tìm kiếm mà chỉ đơn thuần là xem xét theo một hệ thống nào đó tất cả các đối tượng để phát hiện ra đối tượng cần tìm.
- Các kỹ thuật tìm kiếm kinh nghiệm (tìm kiếm heuristic) trong đó chúng ta dựa vào kinh nghiệm và sự hiểu biết của chúng ta về vấn đề cần giải quyết để xây dựng nên hàm đánh giá hướng dẫn sự tìm kiếm.
- Các kỹ thuật tìm kiếm tối ưu.
- Các phương pháp tìm kiếm có đối thủ, tức là các chiến lược tìm kiếm nước đi trong các trò chơi hai người, chẳng hạn cờ vua, cờ tướng, cờ carô.

Chương I

Các chiến lược tìm kiếm mù

Trong chương này, chúng tôi sẽ nghiên cứu các chiến lược tìm kiếm mù (blind search): tìm kiếm theo bề rộng (breadth-first search) và tìm kiếm theo độ sâu (depth-first search). Hiệu quả của các phương pháp tìm kiếm này cũng sẽ được đánh giá.

1.1 Biểu diễn vấn đề trong không gian trạng thái

Một khi chúng ta muốn giải quyết một vấn đề nào đó bằng tìm kiếm, đầu tiên ta phải xác định không gian tìm kiếm. Không gian tìm kiếm bao gồm tất cả các đối tượng mà ta cần quan tâm tìm kiếm. Nó có thể là không gian liên tục, chẳng hạn không gian các véctơ thực n chiều; nó cũng có thể là không gian các đối tượng rời rạc.

Trong mục này ta sẽ xét việc biểu diễn một vấn đề trong không gian trạng thái sao cho việc giải quyết vấn đề được quy về việc tìm kiếm trong không gian trạng thái.

Một phạm vi rộng lớn các vấn đề, đặc biệt các câu đố, các trò chơi, có thể mô tả bằng cách sử dụng khái niệm trạng thái và toán tử (phép biến đổi trạng thái). Chẳng hạn, một khách du lịch có trong tay bản đồ mạng lưới giao thông nối các thành phố trong một vùng lãnh thổ (hình 1.1), du khách đang ở thành phố A và anh ta muốn tìm đường đi tới thăm thành phố B. Trong bài toán này, các thành phố có trong các bản đồ là các trạng thái, thành phố A là trạng thái ban đầu, B là trạng thái kết thúc. Khi đang ở một thành phố, chẳng hạn ở thành phố D anh ta có thể đi theo các con đường để nối tới các thành phố C, F và G. Các con đường nối các thành phố sẽ được biểu diễn bởi các toán tử. Một toán tử biến đổi một trạng thái thành một trạng thái khác. Chẳng hạn, ở trạng thái D sẽ có ba toán tử dẫn trạng thái D tới các trạng thái C, F và G. Vấn đề của du khách bây giờ sẽ là tìm một dãy toán tử để đưa trạng thái ban đầu A tới trạng thái kết thúc B.

Một ví dụ khác, trong trò chơi cờ vua, mỗi cách bố trí các quân trên bàn cờ là một trạng thái. Trạng thái ban đầu là sự sắp xếp các quân lúc bắt đầu cuộc chơi. Mỗi nước đi hợp lệ là một toán tử, nó biến đổi một cảnh huống trên bàn cờ thành một cảnh huống khác.

Như vậy muốn biểu diễn một vấn đề trong không gian trạng thái, ta cần xác định các yếu tố sau:

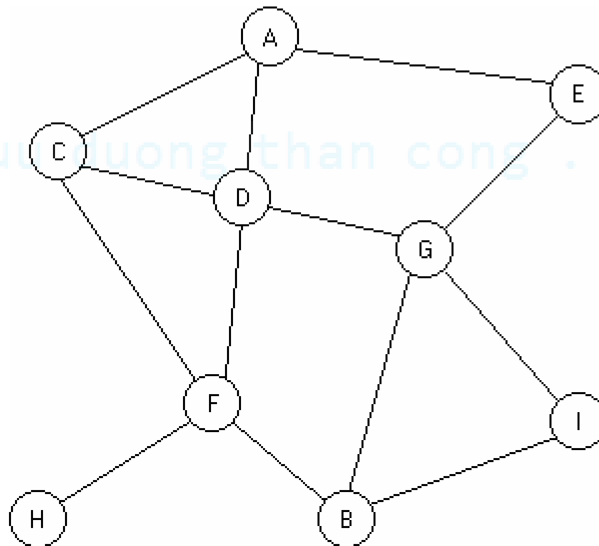
- Trạng thái ban đầu.
- Một tập hợp các toán tử. Trong đó mỗi toán tử mô tả một hành động hoặc một phép biến đổi có thể đưa một trạng thái tới một trạng thái khác.

Tập hợp tất cả các trạng thái có thể đạt tới từ trạng thái ban đầu bằng cách áp dụng một dãy toán tử, lập thành không gian trạng thái của vấn đề.

Ta sẽ ký hiệu không gian trạng thái là U, trạng thái ban đầu là u_0 ($u_0 \in U$). Mỗi toán tử R có thể xem như một ánh xạ R: $U \rightarrow U$. Nói chung R là một ánh xạ không xác định khắp nơi trên U.

- Một tập hợp T các trạng thái kết thúc (trạng thái đích). T là tập con của không gian U. Trong vấn đề của du khách trên, chỉ có một trạng thái đích, đó là thành phố B. Nhưng trong nhiều vấn đề (chẳng hạn các loại cờ) có thể có nhiều trạng thái đích và ta không thể xác định trước được các trạng thái đích. Nói chung trong phần lớn các vấn đề hay, ta chỉ có thể mô tả các trạng thái đích là các trạng thái thỏa mãn một số điều kiện nào đó.

Khi chúng ta biểu diễn một vấn đề thông qua các trạng thái và các toán tử, thì việc tìm nghiệm của bài toán được quy về việc tìm đường đi từ trạng thái ban đầu tới trạng thái đích. (Một đường đi trong không gian trạng thái là một dãy toán tử dẫn một trạng thái tới một trạng thái khác).

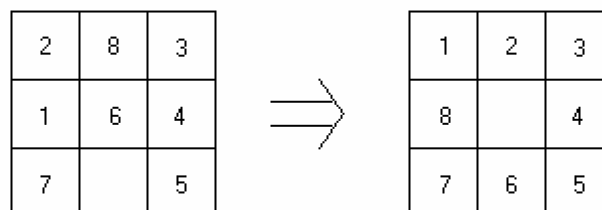


Hình 1.1 Tìm đường đi từ A đến B

Chúng ta có thể biểu diễn không gian trạng thái bằng đồ thị định hướng, trong đó mỗi đỉnh của đồ thị tương ứng với một trạng thái. Nếu có toán tử R biến đổi trạng thái u thành trạng thái v, thì có cung gán nhãn R đi từ đỉnh u tới đỉnh v. Khi đó một đường đi trong không gian trạng thái sẽ là một đường đi trong đồ thị này.

Sau đây chúng ta sẽ xét một số ví dụ về các không gian trạng thái được xây dựng cho một số vấn đề.

Ví dụ 1: Bài toán 8 số. Chúng ta có bảng 3x3 ô và tám quân mang số hiệu từ 1 đến 8 được xếp vào tám ô, còn lại một ô trống, chẳng hạn như trong hình 2 bên



Hình 1.2 Trạng thái ban đầu và trạng thái kết thúc của bài toán số.

trái. Trong trò chơi này, bạn có thể chuyển dịch các quân ở cạch ô trống tới ô trống đó. Vấn đề của bạn là tìm ra một dãy các chuyển dịch để biến đổi cảnh huống ban đầu (hình 1.2 bên trái) thành một cảnh huống xác định nào đó, chẳng hạn cảnh huống trong hình 1.2 bên phải.

Trong bài toán này, trạng thái ban đầu là cảnh huống ở bên trái hình 1.2, còn trạng thái kết thúc ở bên phải hình 1.2. Tương ứng với các quy tắc chuyển dịch các quân, ta có bốn toán tử: ***up*** (đẩy quân lên trên), ***down*** (đẩy quân xuống dưới), ***left*** (đẩy quân sang trái), ***right*** (đẩy quân sang phải). Rõ ràng là, các toán tử này chỉ là các toán tử bộ phận; chẳng hạn, từ trạng thái ban đầu (hình 1.2 bên trái), ta chỉ có thể áp dụng các toán tử ***down, left, right***.

Trong các ví dụ trên việc tìm ra một biểu diễn thích hợp để mô tả các trạng thái của vấn đề là khá dễ dàng và tự nhiên. Song trong nhiều vấn đề việc tìm hiểu được biểu diễn thích hợp cho các trạng thái của vấn đề là hoàn toàn không đơn giản. Việc tìm ra dạng biểu diễn tốt cho các trạng thái đóng vai trò hết sức quan trọng trong quá trình giải quyết một vấn đề. Có thể nói rằng, nếu ta tìm được dạng biểu diễn tốt cho các trạng thái của vấn đề, thì vấn đề hầu như đã được giải quyết.

Ví dụ 2: Vấn đề triệu phú và kẻ cướp. Có ba nhà triệu phú và ba tên cướp ở bên bờ tả ngạn một con sông, cùng một chiếc thuyền chở được một hoặc hai người. Hãy tìm cách đưa mọi người qua sông sao cho không để lại ở bên bờ sông kẻ cướp nhiều hơn triệu phú. Đương nhiên trong bài toán này, các toán tử tương ứng với các hành động chở 1 hoặc 2 người qua sông. Nhưng ở đây ta cần lưu ý rằng, khi hành động xẩy ra (lúc thuyền đang bơi qua sông) thì ở bên bờ sông thuyền vừa dời chỗ, số kẻ cướp không được nhiều hơn số triệu phú. Tiếp theo ta cần quyết định cái gì là trạng thái của vấn đề. ở đây ta không cần phân biệt các nhà triệu phú và các tên cướp, mà chỉ số lượng của họ ở bên bờ sông là quan trọng. Để biểu diễn các trạng thái, ta sử dụng bộ ba (a, b, k), trong đó a là số triệu phú, b là số kẻ cướp ở bên bờ tả ngạn vào các thời điểm mà thuyền ở bờ này hoặc bờ kia, k = 1 nếu thuyền ở bờ tả ngạn và k = 0 nếu thuyền ở bờ hữu ngạn. Như vậy, không gian trạng thái cho bài toán triệu phú và kẻ cướp được xác định như sau:

- Trạng thái ban đầu là (3, 3, 1).
- Các toán tử. Có năm toán tử tương ứng với hành động thuyền chở qua sông 1 triệu phú, hoặc 1 kẻ cướp, hoặc 2 triệu phú, hoặc 2 kẻ cướp, hoặc 1 triệu phú và 1 kẻ cướp.
- Trạng thái kết thúc là (0, 0, 0).

1.2 Các chiến lược tìm kiếm

Như ta đã thấy trong mục 1.1, để giải quyết một vấn đề bằng tìm kiếm trong không gian trạng thái, đầu tiên ta cần tìm dạng thích hợp mô tả các trạng thái cảu vấn đề. Sau đó cần xác định:

- Trạng thái ban đầu.

 Tập các toán tử.

 Tập T các trạng thái kết thúc. (T có thể không được xác định cụ thể gồm các trạng thái nào mà chỉ được chỉ định bởi một số điều kiện nào đó).

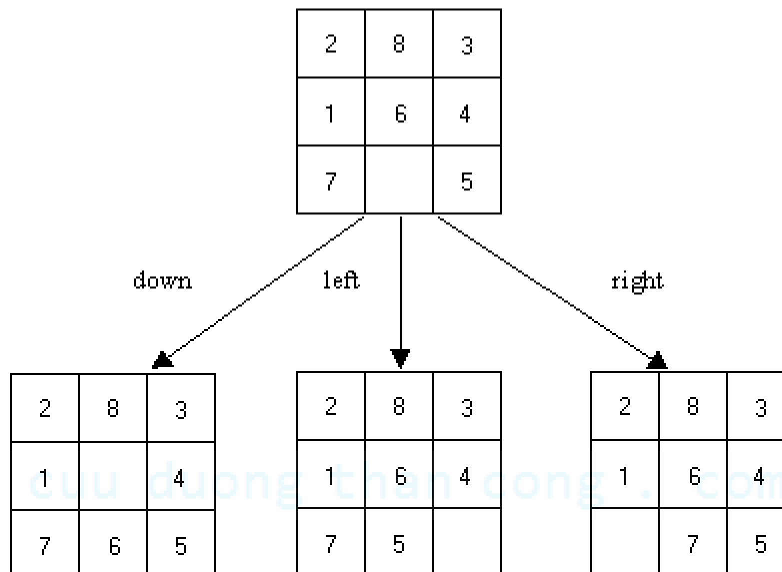
Giả sử u là một trạng thái nào đó và R là một toán tử biến đổi u thành v. Ta sẽ gọi v là trạng thái kề u, hoặc v được sinh ra từ trạng thái u bởi toán tử R. Quá trình áp dụng các toán tử để sinh ra các trạng thái kề u được gọi là phát triển trạng thái u. Chẳng hạn, trong bài toán toán số, phát triển trạng thái ban đầu (hình 2 bên trái), ta nhận được ba trạng thái kề (hình 1.3).

Khi chúng ta biểu diễn một vấn đề cần giải quyết thông qua các trạng thái và các toán tử thì việc tìm lời giải của vấn đề được quy về việc tìm đường đi từ trạng thái ban đầu tới một trạng thái kết thúc nào đó.

Có thể phân các chiến lược tìm kiếm thành hai loại:

 Các chiến lược tìm kiếm mù. Trong các chiến lược tìm kiếm này, không có một sự hướng dẫn nào cho sự tìm kiếm, mà ta chỉ phát triển các trạng thái ban đầu cho tới khi gặp một trạng thái đích nào đó. Có hai kỹ thuật tìm kiếm mù, đó là tìm kiếm theo bề rộng và tìm kiếm theo độ sâu.

Tư tưởng của tìm kiếm theo bề rộng là các trạng thái được phát triển theo thứ tự mà chúng được sinh ra, tức là trạng thái nào được sinh ra trước sẽ được phát triển trước.



Hình 1.3 Phát triển một trạng thái

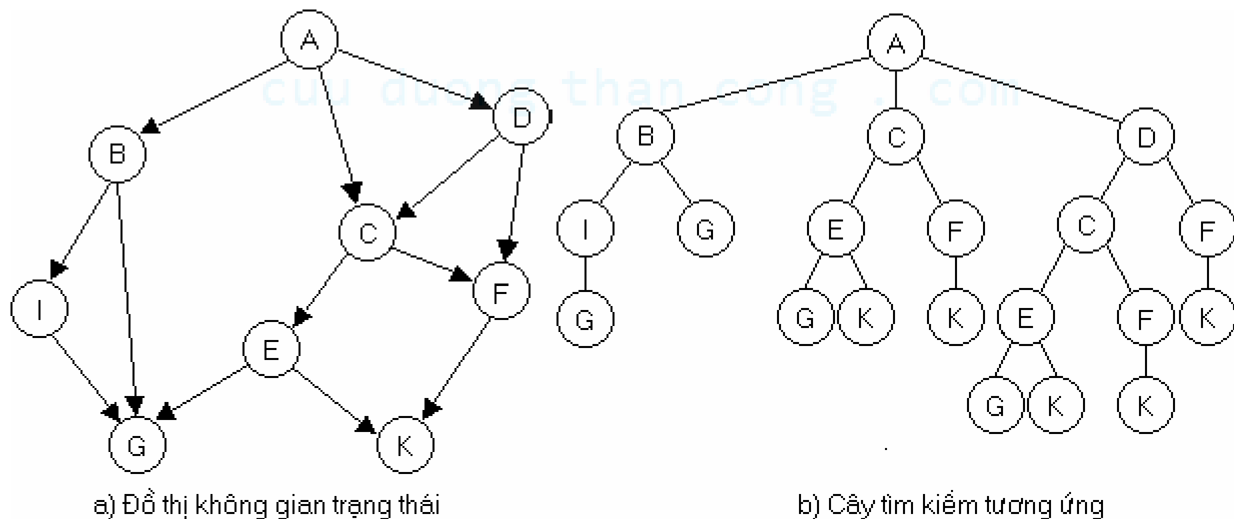
Trong nhiều vấn đề, dù chúng ta phát triển các trạng thái theo hệ thống nào (theo bề rộng hoặc theo độ sâu) thì số lượng các trạng thái được sinh ra trước khi ta gặp trạng thái đích thường là cực kỳ lớn. Do đó các thuật toán tìm kiếm mù kém hiệu quả, đòi hỏi rất nhiều không gian và thời gian. Trong thực tế, nhiều vấn đề không thể giải quyết được bằng tìm kiếm mù.

 Tìm kiếm kinh nghiệm (tìm kiếm heuristic). Trong rất nhiều vấn đề, chúng ta có thể dựa vào sự hiểu biết của chúng ta về vấn đề, dựa vào kinh nghiệm, trực giác, để đánh giá các trạng thái. Sử dụng sự đánh giá các trạng thái để hướng dẫn sự tìm kiếm: trong quá trình phát triển các trạng thái, ta sẽ chọn trong số các trạng thái chờ phát triển, trạng thái được đánh giá là tốt nhất để phát triển. Do đó tốc độ tìm kiếm sẽ nhanh hơn. Các phương pháp tìm kiếm dựa vào sự đánh giá các trạng thái để hướng dẫn sự tìm kiếm gọi chung là các phương pháp tìm kiếm kinh nghiệm.

Như vậy chiến lược tìm kiếm được xác định bởi chiến lược chọn trạng thái để phát triển ở mỗi bước. Trong tìm kiếm mù, ta chọn trạng thái để phát triển theo thứ tự mà đúng được sinh ra; còn trong tìm kiếm kinh nghiệm ta chọn trạng thái dựa vào sự đánh giá các trạng thái.

Cây tìm kiếm

Chúng ta có thể nghĩ đến quá trình tìm kiếm như quá trình xây dựng ***cây tìm kiếm***. Cây tìm kiếm là cây mà các đỉnh được gắn bởi các trạng thái của không gian trạng thái. Gốc của cây tìm kiếm tương ứng với trạng thái ban đầu. Nếu một đỉnh ứng với trạng thái u, thì các đỉnh con của nó ứng với các trạng thái v kề u. Hình



Hình 1.4

1.4a là đồ thị biểu diễn một không gian trạng thái với trạng thái ban đầu là A, hình 1.4b là cây tìm kiếm tương ứng với không gian trạng thái đó.

Mỗi chiến lược tìm kiếm trong không gian trạng thái tương ứng với một phương pháp xây dựng cây tìm kiếm. Quá trình xây dựng cây bắt đầu từ cây chỉ có một đỉnh là trạng thái ban đầu. Giả sử tới một bước nào đó trong chiến lược tìm kiếm, ta đã xây dựng được một cây nào đó, các lá của cây tương ứng với các trạng thái chưa được phát triển. Bước tiếp theo phụ thuộc vào chiến lược tìm kiếm mà một đỉnh nào đó trong các lá được chọn để phát triển. Khi phát triển đỉnh đó, cây tìm kiếm được mở rộng bằng cách thêm vào các đỉnh con của đỉnh đó. Kỹ thuật

tìm kiếm theo bề rộng (theo độ sâu) tương ứng với phương pháp xây dựng cây tìm kiếm theo bề rộng (theo độ sâu).

1.3 Các chiến lược tìm kiếm mù

Trong mục này chúng ta sẽ trình bày hai chiến lược tìm kiếm mù: tìm kiếm theo bề rộng và tìm kiếm theo độ sâu. Trong tìm kiếm theo bề rộng, tại mỗi bước ta sẽ chọn trạng thái để phát triển là trạng thái được sinh ra trước các trạng thái chờ phát triển khác. Còn trong tìm kiếm theo độ sâu, trạng thái được chọn để phát triển là trạng thái được sinh ra sau cùng trong số các trạng thái chờ phát triển.

Chúng ta sử dụng danh sách L để lưu các trạng thái đã được sinh ra và chờ được phát triển. Mục tiêu của tìm kiếm trong không gian trạng thái là tìm đường đi từ trạng thái ban đầu tới trạng thái đích, do đó ta cần lưu lại vết của đường đi. Ta có thể sử dụng hàm father để lưu lại cha của mỗi đỉnh trên đường đi, ***father***(v) = u nếu cha của đỉnh v là u.

1.3.1 Tìm kiếm theo bề rộng

Thuật toán tìm kiếm theo bề rộng được mô tả bởi thủ tục sau:

procedure *Breadth_First_Search*;

begin

1. Khởi tạo danh sách L chỉ chứa trạng thái ban đầu;

2. **loop do**

2.1 **if** *L rỗng* **then**

{*thông báo tìm kiếm thất bại***; stop};**

*2.2 Loại trạng thái u ở đầu danh sách L***;**

2.3 **if** *u là trạng thái kết thúc* **then**

{*thông báo tìm kiếm thành công***; stop};**

2.4 **for** *mỗi trạng thái v kề u* **do {**

*Đặt v vào cuối danh sách L***;**

*father(v) <- u***}**

end;

Chúng ta có một số nhận xét sau đây về thuật toán tìm kiếm theo bề rộng:

- Trong tìm kiếm theo bề rộng, trạng thái nào được sinh ra trước sẽ được phát triển trước, do đó danh sách L được xử lý như hàng đợi. Trong bước 2.3, ta cần kiểm tra xem u có là trạng thái kết thúc hay không. Nói chung các trạng thái kết thúc được xác định bởi một số điều kiện nào đó, khi đó ta cần kiểm tra xem u có thỏa mãn các điều kiện đó hay không.

- Nếu bài toán có nghiệm (tồn tại đường đi từ trạng thái ban đầu tới trạng thái đích), thì thuật toán tìm kiếm theo bề rộng sẽ tìm ra nghiệm, đồng thời đường đi

tìm được sẽ là ngắn nhất. Trong trường hợp bài toán vô nghiệm và không gian trạng thái hữu hạn, thuật toán sẽ dừng và cho thông báo vô nghiệm.

Đánh giá tìm kiếm theo bề rộng

Bây giờ ta đánh giá thời gian và bộ nhớ mà tìm kiếm theo bề rộng đòi hỏi. Giả sử rằng, mỗi trạng thái khi được phát triển sẽ sinh ra b trạng thái kề. Ta sẽ gọi b là ***nhân tố nhánh***. Giả sử rằng, nghiệm của bài toán là đường đi có độ dài d. Bởi nhiều nghiệm có thể được tìm ra tại một đỉnh bất kỳ ở mức d của cây tìm kiếm, do đó số đỉnh cần xem xét để tìm ra nghiệm là:

$$\mathbf{1 + b + b^2 + ... + b^{d-1} + k}$$

Trong đó k có thể là 1, 2, ..., b^d. Do đó số lớn nhất các đỉnh cần xem xét là:

$$\mathbf{1 + b + b^2 + ... + b^d}$$

Như vậy, độ phức tạp thời gian của thuật toán tìm kiếm theo bề rộng là $O(b^d)$. Độ phức tạp không gian cũng là $O(b^d)$, bởi vì ta cần lưu vào danh sách L tất cả các đỉnh của cây tìm kiếm ở mức d, số các đỉnh này là b^d.

Để thấy rõ tìm kiếm theo bề rộng đòi hỏi thời gian và không gian lớn tới mức nào, ta xét trường hợp nhân tố nhánh b = 10 và độ sâu d thay đổi. Giả sử để phát hiện và kiểm tra 1000 trạng thái cần 1 giây, và lưu giữ 1 trạng thái cần 100 bytes. Khi đó thời gian và không gian mà thuật toán đòi hỏi được cho trong bảng sau:

Độ sâu d	Thời gian	Không gian
4	11 giây	1 megabyte
6	18 giây	111 megabytes
8	31 giờ	11 gigabytes
10	128 ngày	1 terabyte
12	35 năm	111 terabytes
14	3500 năm	11.111 terabytes

1.3.2 Tìm kiếm theo độ sâu

Như ta đã biết, tư tưởng của chiến lược tìm kiếm theo độ sâu là, tại mỗi bước trạng thái được chọn để phát triển là trạng thái được sinh ra sau cùng trong số các trạng thái chờ phát triển. Do đó thuật toán tìm kiếm theo độ sâu là hoàn toàn tương tự như thuật toán tìm kiếm theo bề rộng, chỉ có một điều khác là, ta xử lý danh sách L các trạng thái chờ phát triển không phải như hàng đợi mà như ngăn xếp. Cụ thể là trong bước 2.4 của thuật toán tìm kiếm theo bề rộng, ta cần sửa lại là "Đặt v vào ***đầu*** danh sách L".

Sau đây chúng ta sẽ đưa ra các nhận xét so sánh hai chiến lược tìm kiếm mù:

 Thuật toán tìm kiếm theo bề rộng luôn luôn tìm ra nghiệm nếu bài toán có nghiệm. Song không phải với bất kỳ bài toán có nghiệm nào thuật toán tìm kiếm theo độ sâu cũng tìm ra nghiệm! Nếu bài toán có nghiệm và không gian trạng thái hữu hạn, thì thuật toán tìm kiếm theo độ sâu sẽ tìm ra nghiệm. Tuy nhiên, trong trường hợp không gian trạng thái vô hạn, thì có thể nó không tìm ra nghiệm, lý do là ta luôn luôn đi xuống theo độ sâu, nếu ta đi theo một nhánh vô hạn mà nghiệm không nằm trên nhánh đó thì thuật toán sẽ không dừng. Do đó người ta khuyên rằng, không nên áp dụng tìm kiếm theo dộ sâu cho các bài toán có cây tìm kiếm chứa các nhánh vô hạn.

 Độ phức tạp của thuật toán tìm kiếm theo độ sâu.

Giả sử rằng, nghiệm của bài toán là đường đi có độ dài d, cây tìm kiếm có nhân tố nhánh là b và có chiều cao là d. Có thể xẩy ra, nghiệm là đỉnh ngoài cùng bên phải trên mức d của cây tìm kiếm, do đó độ phức tạp thời gian của tìm kiếm theo độ sâu trong trường hợp xấu nhất là $O(b^d)$, tức là cũng như tìm kiếm theo bề rộng. Tuy nhiên, trên thực tế đối với nhiều bài toán, tìm kiếm theo độ sâu thực sự nhanh hơn tìm kiếm theo bề rộng. Lý do là tìm kiếm theo bề rộng phải xem xét toàn bộ cây tìm kiếm tới mức d-1, rồi mới xem xét các đỉnh ở mức d. Còn trong tìm kiếm theo độ sâu, có thể ta chỉ cần xem xét một bộ phận nhỏ của cây tìm kiếm thì đã tìm ra nghiệm.

Để đánh giá độ phức tạp không gian của tìm kiếm theo độ sâu ta có nhận xét rằng, khi ta phát triển một đỉnh u trên cây tìm kiếm theo độ sâu, ta chỉ cần lưu các đỉnh chưa được phát triển mà chúng là các đỉnh con của các đỉnh nằm trên đường đi từ gốc tới đỉnh u. Như vậy đối với cây tìm kiếm có nhân tố nhánh b và độ sâu lớn nhất là d, ta chỉ cần lưu ít hơn db đỉnh. Do đó độ phức tạp không gian của tìm kiếm theo độ sâu là O(db), trong khi đó tìm kiếm theo bề rộng đòi hỏi không gian nhớ $O(b^d)$!

1.3.3 Các trạng thái lặp

Như ta thấy trong mục 1.2, cây tìm kiếm có thể chứa nhiều đỉnh ứng với cùng một trạng thái, các trạng thái này được gọi là trạng thái lặp. Chẳng hạn, trong cây tìm kiếm hình 4b, các trạng thái C, E, F là các trạng thái lặp. Trong đồ thị biểu diễn không gian trạng thái, các trạng thái lặp ứng với các đỉnh có nhiều đường đi dẫn tới nó từ trạng thái ban đầu. Nếu đồ thị có chu trình thì cây tìm kiếm sẽ chứa các nhánh với một số đỉnh lập lại vô hạn lần. Trong các thuật toán tìm kiếm sẽ lãng phí rất nhiều thời gian để phát triển lại các trạng thái mà ta đã gặp và đã phát triển. Vì vậy trong quá trình tìm kiếm ta cần tránh phát sinh ra các trạng thái mà ta đã phát triển. Chúng ta có thể áp dụng một trong các giải pháp sau đây:

1. Khi phát triển đỉnh u, không sinh ra các đỉnh trùng với cha của u.
2. Khi phát triển đỉnh u, không sinh ra các đỉnh trùng với một đỉnh nào đó nằm trên đường đi dẫn tới u.

3. Không sinh ra các đỉnh mà nó đã được sinh ra, tức là chỉ sinh ra các đỉnh mới.

Hai giải pháp đầu dễ cài đặt và không tốn nhiều không gian nhớ, tuy nhiên các giải pháp này không tránh được hết các trạng thái lặp.

Để thực hiện giải pháp thứ 3 ta cần lưu các trạng thái đã phát triển vào tập Q, lưu các trạng thái chờ phát triển vào danh sách L. Đương nhiên, trạng thái v lần đầu được sinh ra nếu nó không có trong Q và L. Việc lưu các trạng thái đã phát triển và kiểm tra xem một trạng thái có phải lần đầu được sinh ra không đòi hỏi rất nhiều không gian và thời gian. Chúng ta có thể cài đặt tập Q bởi bảng băm (xem [ ]).

1.3.4 Tìm kiếm sâu lặp

Như chúng ta đã nhận xét, nếu cây tìm kiếm chứa nhánh vô hạn, khi sử dụng tìm kiếm theo độ sâu, ta có thể mắc kẹt ở nhánh đó và không tìm ra nghiệm. Để khắc phục hoàn cảnh đó, ta tìm kiếm theo độ sâu chỉ tới mức d nào đó; nếu không tìm ra nghiệm, ta tăng độ sâu lên d+1 và lại tìm kiếm theo độ sâu tới mức d+1. Quá trình trên được lặp lại với d lần lượt là 1, 2, ... dến một độ sâu max nào đó. Như vậy, thuật toán tìm kiếm sâu lặp (iterative deepening search) sẽ sử dụng thủ tục tìm kiếm sâu hạn chế (depth_limited search) như thủ tục con. Đó là thủ tục tìm kiếm theo độ sâu, nhưng chỉ đi tới độ sâu d nào đó rồi quay lên.

Trong thủ tục tìm kiếm sâu hạn chế, d là tham số độ sâu, hàm depth ghi lại độ sâu của mỗi đỉnh

procedure *Depth_Limited_Search(d)***;**

begin

1. Khởi tạo danh sách L chỉ chứa trạng thái ban đầu u_0**;**

  depth(u_0) ← *0***;**

2. **loop do**

  2.1 **if** *L rỗng* **then**

    {*thông báo thất bại***; stop};**

  *2.2 Loại trạng thái u ở đầu danh sách L***;**

  2.3 **if** *u là trạng thái kết thúc* **then**

    {*thông báo thành công***; stop};**

  2.4 **if** *depth(u) <= d* **then**

    for *mỗi trạng thái v kề u* **do**

      {*Đặt v vào đầu danh sách L***;**

      depth(v) ← *depth(u) + 1***};**

end;

```
procedure Depth_Deepening_Search;
begin
for d ← 0 to max do
        {Depth_Limited_Search(d);
         if thành công then exit}
end;
```

Kỹ thuật tìm kiếm sâu lặp kết hợp được các ưu điểm của tìm kiếm theo bề rộng và tìm kiếm theo độ sâu. Chúng ta có một số nhận xét sau:

- Cũng như tìm kiếm theo bề rộng, tìm kiếm sâu lặp luôn luôn tìm ra nghiệm (nếu bài toán có nghiệm), miễn là ta chọn độ sâu mã đủ lớn.
- Tìm kiếm sâu lặp chỉ cần không gian nhớ như tìm kiếm theo độ sâu.
- Trong tìm kiếm sâu lặp, ta phải phát triển lặp lại nhiều lần cùng một trạng thái. Điều đó làm cho ta có cảm giác rằng, tìm kiếm sâu lặp lãng phí nhiều thời gian. Thực ra thời gian tiêu tốn cho phát triển lặp lại các trạng thái là không đáng kể so với thời gian tìm kiếm theo bề rộng. Thật vậy, mỗi lần gọi thủ tục tìm kiếm sâu hạn chế tới mức d, nếu cây tìm kiếm có nhân tố nhánh là b, thì số đỉnh cần phát triển là:

$$\mathbf{1 + b + b^2 + ... + b^d}$$

Nếu nghiệm ở độ sâu d, thì trong tìm kiếm sâu lặp, ta phải gọi thủ tục tìm kiếm sâu hạn chế với độ sâu lần lượt là 0, 1, 2, ..., d. Do đó các đỉnh ở mức 1 phải phát triển lặp d lần, các đỉnh ở mức 2 lặp d-1 lần, ..., các đỉnh ở mức d lặp 1 lần. Như vậy tổng số đỉnh cần phát triển trong tìm kiếm sâu lặp là:

$$\mathbf{(d+1)1 + db + (d-1)b^2 + ... + 2b^{d-1} + 1b^d}$$

Do đó thời gian tìm kiếm sâu lặp là $O(b^d)$.

Tóm lại, tìm kiếm sâu lặp có độ phức tạp thời gian là $O(b^d)$ (như tìm kiếm theo bề rộng), và có độ phức tạp không gian là O(biểu diễn) (như tìm kiếm theo độ sâu). Nói chung, chúng ta nên áp dụng tìm kiếm sâu lặp cho các vấn đề có không gian trạng thái lớn và độ sâu của nghiệm không biết trước.

1.4 Quy vấn đề về các vấn đề con. Tìm kiếm trên đồ thị và/hoặc.

1.4.1 Quy vấn đề về các vấn đề con:

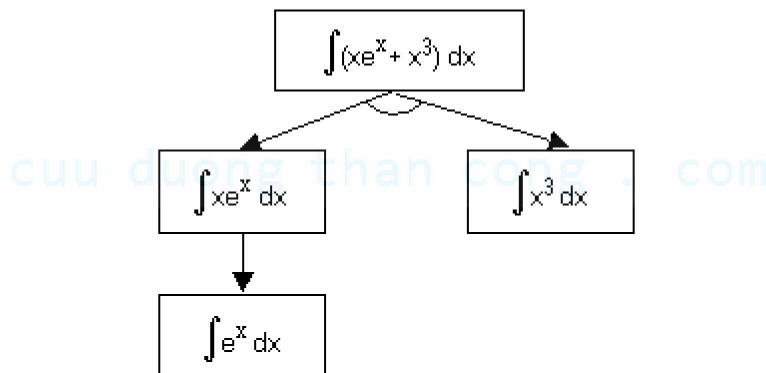
Trong mục 1.1, chúng ta đã nghiên cứu việc biểu diễn vấn đề thông qua các trạng thái và các toán tử. Khi đó việc tìm nghiệm của vấn đề được quy về việc tìm đường trong không gian trạng thái. Trong mục này chúng ta sẽ nghiên cứu một phương pháp luận khác để giải quyết vấn đề, dựa trên việc quy vấn đề về các vấn đề con. Quy vấn đề về các vấn đề con (còn gọi là rút gọn vấn đề) là một phương pháp được sử dụng rộng rãi nhất để giải quyết các vấn đề. Trong đời sống hàng ngày, cũng như trong khoa học kỹ thuật, mỗi khi gặp một vấn đề cần giải quyết, ta

vẫn thường cố gắng tìm cách đưa nó về các vấn đề đơn giản hơn. Quá trình rút gọn vấn đề sẽ được tiếp tục cho tới khi ta dẫn tới các vấn đề con có thể giải quyết được dễ dàng. Sau đây chúng ta xét một số vấn đề.

Vấn đề tính tích phân bất định

Giả sử ta cần tính một tích phân bất định, chẳng hạn $\int (xe^x + x^3)\, dx$. Quá trình chúng ta vẫn thường làm để tính tích phân bất định là như sau. Sử dụng các quy tắc tính tích phân (quy tắc tính tích phân của một tổng, quy tắc tính tích phân từng phần...), sử dụng các phép biến đổi biến số, các phép biến đổi các hàm (chẳng hạn, các phép biến đổi lượng giác),... để đưa tích phân cần tính về tích phân của các hàm số sơ cấp mà chúng ta đã biết cách tính. Chẳng hạn, đối với tích phân $\int (xe^x + x^3)\, dx$, áp dụng quy tắc tích phân của tổng ta đưa về hai tích phân $\int xe^x dx$ và $\int x^3 dx$. áp dụng quy tắc tích phân từng phần ta đưa tích phân $\int xe^x dx$ về tích phân $\int e^x dx$. Quá trình trên có thể biểu diễn bởi đồ thị trong hình 1.5.

Các tích phân $\int e^x dx$ và $\int x^3 dx$ là các tích phân cơ bản đã có trong bảng tích phân. Kết hợp các kết quả của các tích phân cơ bản, ta nhận được kết quả của tích



Hình 1.5 Quy một số tích phân về các tích phân cơ bản.

phân đã cho.

Chúng ta có thể biểu diễn việc quy một vấn đề về các vấn đề con cơ bởi các trạng thái và các toán tử. ở đây, bài toán cần giải là trạng thái ban đầu. Mỗi cách quy bài toán về các bài toán con được biểu diễn bởi một toán tử, toán tử A→B, C biểu diễn việc quy bài toán A về hai bài toán B và C. Chẳng hạn, đối với bài toán tính tích phân bất định, ta có thể xác định các toán tử dạng:

$\int (f_1 + f_2)\, dx \rightarrow \int f_1\, dx, \int f_2\, dx$ và $\int u\, dv \rightarrow \int v\, du$

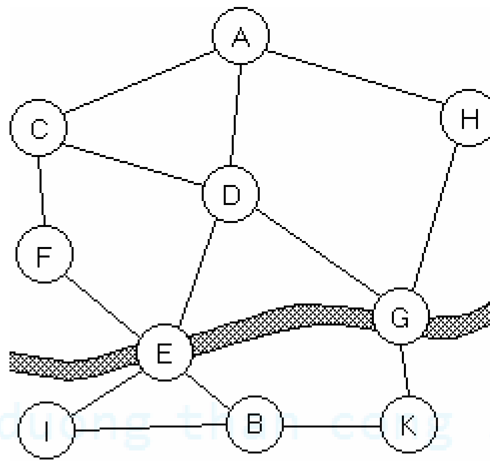
Các trạng thái kết thúc là các bài toán sơ cấp (các bài toán đã biết cách giải). Chẳng hạn, trong bài toán tính tích phân, các tích phân cơ bản là các trạng thái kết thúc. Một điều cần lưu ý là, trong không gian trạng thái biểu diễn việc quy vấn đề về các vấn đề con, các toán tử có thể là đa trị, nó biến đổi một trạng thái thành nhiều trạng thái khác.

Vấn đề tìm đường đi trên bản đồ giao thông

Bài toán này đã được phát triển như bài toán tìm đường đi trong không gian trạng thái (xem 1.1), trong đó mỗi trạng thái ứng với một thành phố, mỗi toán tử ứng với một con đường nối, nối thành phố này với thành phố khác. Bây giờ ta đưa ra một cách biểu diễn khác dựa trên việc quy vấn đề về các vấn đề con. Giả sử ta có bản đồ giao thông trong một vùng lãnh thổ (xem hình 1.6). Giả sử ta cần tìm đường đi từ thành phố A tới thành phố B. Có con sông chảy qua hai thành phố E và G và có cầu qua sông ở mỗi thành phố đó. Mọi đường đi từ A đến B chỉ có thể qua E hoặc G. Như vậy bài toán tìm đường đi từ A đến B được quy về:

1) Bài toán tìm đường đi từ A đến B qua E (hoặc)

2) Bài toán tìm đường đi từ A đến b qua G.



Hình 1.6

Mỗi một trong hai bài toán trên lại có thể phân nhỏ như sau

1) Bài toán tìm đường đi từ A đến B qua E được quy về:

  1.1 Tìm đường đi từ A đến E (và)

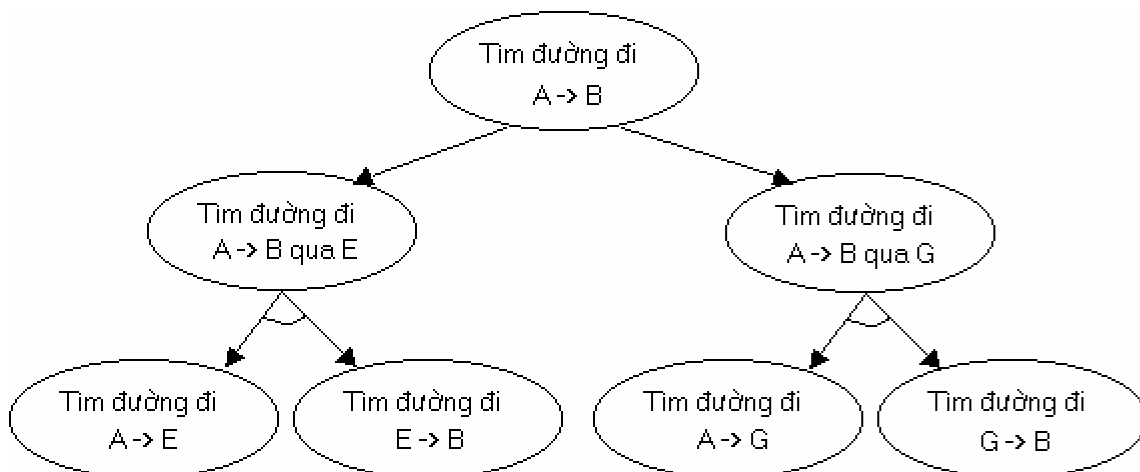
  1.2 Tìm đường đi từ E đến B.

2) Bài toán tìm đường đi từ A đến B qua G được quy về:

  2.1 Tìm đường đi từ A đến G (và)

  2.2 Tìm đường đi từ G đến B.

Quá trình rút gọn vấn đề như trên có thể biểu diễn dưới dạng đồ thị (đồ thị và/hoặc) trong hình 1.7. ở đây mỗi bài toán tìm đường đi từ một thành phố tới một thành phố khác ứng với một trạng thái. Các trạng thái kết thúc là các trạng thái ứng với các bài toán tìm đường đi, chẳng hạn từ A đến C, hoặc từ D đến E, bởi vì đã có đường nối A với C, nối D với E.

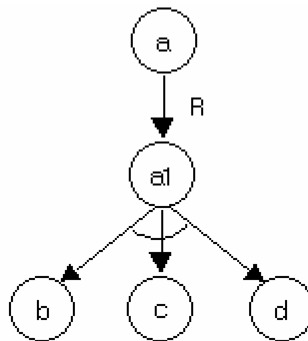


Hình 1.7 Đồ thị và/hoặc của vấn đề tìm đường đi.

1.4.2 Đồ thị và/hoặc

Không gian trạng thái mô tả việc quy vấn đề về các vấn đề con có thể biểu diễn dưới dạng đồ thị định hướng đặc biệt được gọi là đồ thị và/hoặc. Đồ thị này được xây dựng như sau:

Mỗi bài toán ứng với một đỉnh của đồ thị. Nếu có một toán tử quy một bài toán về một bài toán khác, chẳng hạn R : a →b, thì trong đồ thị sẽ có cung gán nhãn đi từ đỉnh a tới đỉnh b. Đối với mỗi toán tử quy một bài toán về một số bài toán con, chẳng hạn R : a →b, c, d ta đưa vào một đỉnh mới a_1, đỉnh này biểu diễn tập các bài toán con {b, c, d} và toán tử R : a →b, c, d được biểu diễn bởi đồ thị



Hình 1.8 Đồ thị biểu diễn toán tử R : a->b, c, d

hình 1.8.

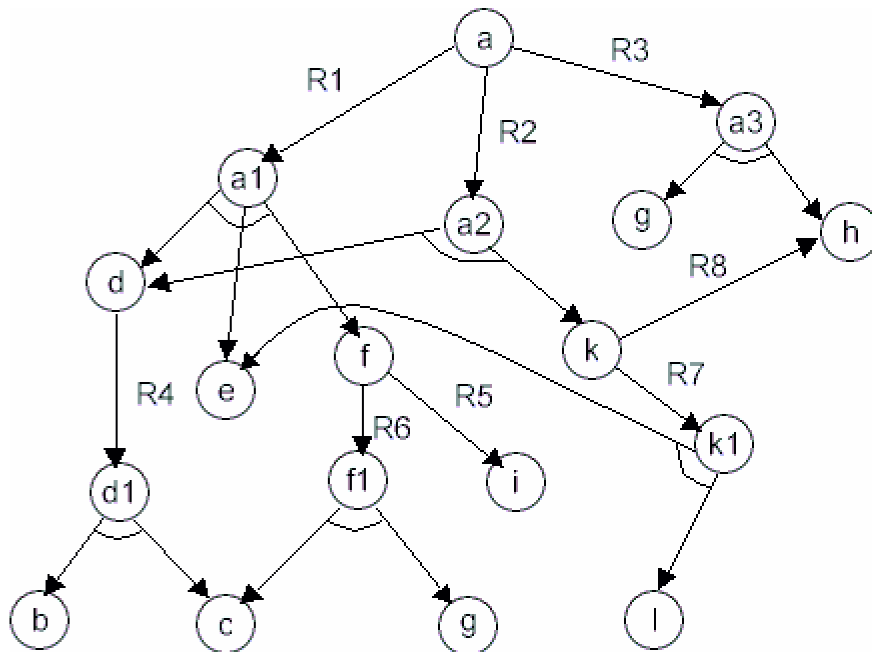
Ví dụ: Giả sử chúng ta có không gian trạng thái sau:

- Trạng thái ban đầu (bài toán cần giải) là a.
- Tập các toán tử quy gồm:

R_1 : a →d, e, f
R_2 : a →d, k
R_3 : a →g, h
R_4 : d →b, c
R_5 : f →i
R_6 : f →c, j
R_7 : k →e, l

$R_8 : k \rightarrow h$

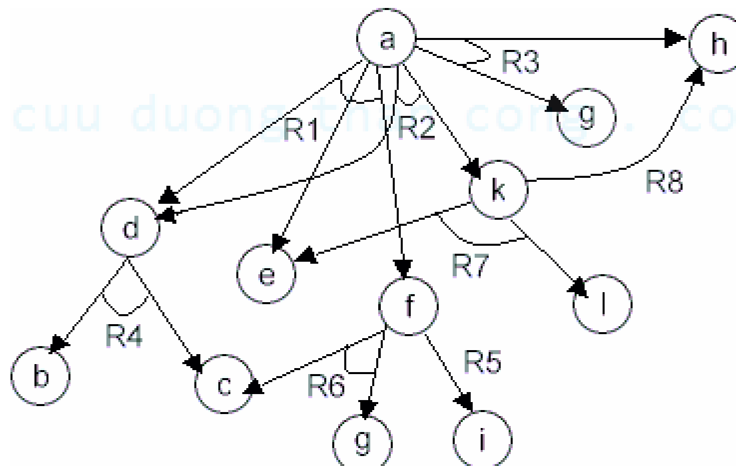
- Tập các trạng thái kết thúc (các bài toán sơ cấp) là T = {b, c, e, j, l}.



Hình 1.9 Một đồ thị và/hoặc

Không gian trạng thái trên có thể biểu diễn bởi đồ thị và/hoặc trong hình 1.9. Trong đồ thị đó, các đỉnh, chẳng hạn a_1, a_2, a_3 được gọi là đỉnh ***và***, các đỉnh chẳng hạn a, f, k được gọi là đỉnh ***hoặc***. Lý do là, đỉnh a_1 biểu diễn tập các bài toán {d, e, f} và a_1 được giải quyết nếu d và e và f được giải quyết. Còn tại đỉnh a, ta có các toán tử R_1, R_2, R_3 quy bài toán a về các bài toán con khác nhau, do đó a được giải quyết nếu hoặc a_1 = {d, e, f}, hoặc a_2 = {d, k}, hoặc a_3 = {g, h} được giải quyết.

Người ta thường sử dụng đồ thị và/hoặc ở dạng rút gọn. Chẳng hạn, đồ thị và/hoặc trong hình 1.9 có thể rút gọn thành đồ thị trong hình 1.10. Trong đồ thị rút gọn này, ta sẽ nói chẳng hạn d, e, f là các đỉnh kề đỉnh a theo toán tử R_1, còn d, k là các đỉnh kề a theo toán tử R_2.



Hình 1.10 Đồ thị rút gọn của đồ thị trong hình 1.9

Khi đã có các toán tử rút gọn vấn đề, thì bằng cách áp dụng liên tiếp các toán tử, ta có thể đưa bài toán cần giải về một tập các bài toán con. Chẳng hạn, trong ví dụ trên nếu ta áp dụng các toán tử R_1, R_4, R_6, ta sẽ quy bài toán a về tập các bài toán con {b, c, e, f}, tất cả các bài toán con này đều là sơ cấp. Từ các toán tử R_1, R_4 và R_6 ta xây dựng được một cây trong hình 1.11a, cây này được gọi là cây nghiệm. Cây nghiệm được định nghĩa như sau:

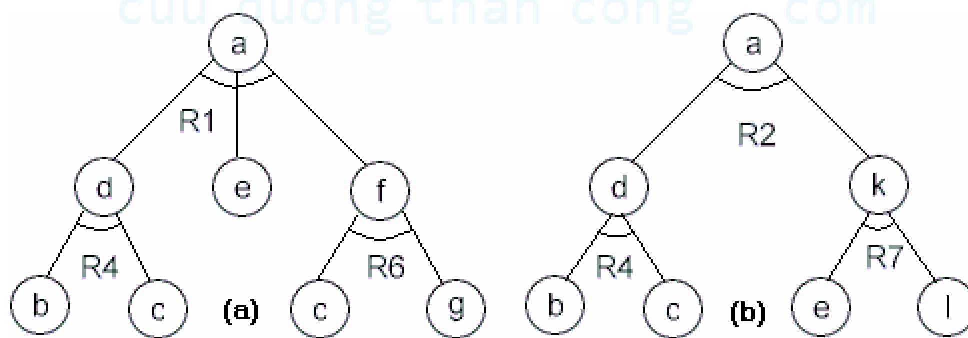
Cây nghiệm là một cây, trong đó:

- Gốc của cây ứng với bài toán cần giải.
- Tất cả các lá của cây là các đỉnh kết thúc (đỉnh ứng với các bài toán sơ cấp).
- Nếu u là đỉnh trong của cây, thì các đỉnh con của u là các đỉnh kề u theo một toán tử nào đó.

Các đỉnh của đồ thị và/hoặc sẽ được gắn nhãn giải được hoặc không giải được.

Các đỉnh ***giải được*** được xác định đệ quy như sau:

- Các đỉnh kết thúc là các đỉnh ***giải được***.
- Nếu u không phải là đỉnh kết thúc, nhưng có một toán tử R sao cho tất cả các đỉnh kề u theo R đều giải được thì u ***giải được***.



Hình 1.11 Các cây nghiệm

Các đỉnh ***không giải được*** được xác định đệ quy như sau:

- Các đỉnh không phải là đỉnh kết thúc và không có đỉnh kề, là các đỉnh ***không giải được***.
- Nếu u không phải là đỉnh kết thúc và với mọi toán tử R áp dụng được tại u đều có một đỉnh v kề u theo R không giải được, thì u ***không giải được***.

Ta có nhận xét rằng, nếu bài toán a ***giải được*** thì sẽ có một cây nghiệm gốc a, và ngược lại nếu có một cây nghiệm gốc a thì a ***giải được***. Hiển nhiên là, một bài toán giải được có thể có nhiều cây nghiệm, mỗi cây nghiệm biểu diễn một cách giải bài toán đó. Chẳng hạn trong ví dụ đã nêu, bài toán a có hai cây nghiệm trong hình 1.11.

Thứ tự giải các bài toán con trong một cây nghiệm là như sau. Bài toán ứng với đỉnh u chỉ được giải sau khi tất cả các bài toán ứng với các đỉnh con của u đã được giải. Chẳng hạn, với cây nghiệm trong hình 1.11a, thứ tự giải các bài toán có thể là b, c, d, j, f, e, a. ta có thể sử dụng thủ tục sắp xếp topo (xem [ ]) để sắp xếp thứ tự các bài toán trong một cây nghiệm. Đương nhiên ta cũng có thể giải quyết đồng thời các bài toán con ở cùng một mức trong cây nghiệm.

Vấn đề của chúng ta bây giờ là, tìm kiếm trên đồ thị và/hoặc để xác định được đỉnh ứng với bài toán ban đầu là giải được hay không giải được, và nếu nó giải được thì xây dựng một cây nghiệm cho nó.

1.4.3 Tìm kiếm trên đồ thị và/hoặc

Ta sẽ sử dụng kỹ thuật tìm kiếm theo độ sâu trên đồ thị và/hoặc để đánh dấu các đỉnh. Các đỉnh sẽ được đánh dấu giải được hoặc không giải được theo định nghĩa đệ quy về đỉnh giải được và không giải được. Xuất phát từ đỉnh ứng với bài toán ban đầu, đi xuống theo độ sâu, nếu gặp đỉnh u là đỉnh kết thúc thì nó được đánh dấu giải được. Nếu gặp đỉnh u không phải là đỉnh kết thúc và từ u không đi tiếp được, thì u được đánh dấu không giải được. Khi đi tới đỉnh u, thì từ u ta lần lượt đi xuống các đỉnh v kề u theo một toán tử R nào đó. Nếu đánh dấu được một đỉnh v không giải được thì không cần đi tiếp xuống các đỉnh v còn lại. Tiếp tục đi xuống các đỉnh kề u theo một toán tử khác. Nếu tất cả các đỉnh kề u theo một toán tử nào đó được đánh dấu giải được thì u sẽ được đánh dấu giải được và quay lên cha của u. Còn nếu từ u đi xuống các đỉnh kề nó theo mọi toán tử đều gặp các đỉnh kề được đánh dấu không giải được, thì u được đánh dấu không giải được và quay lên cha của u.

Ta sẽ biểu diễn thủ tục tìm kiếm theo độ sâu và đánh dấu các đỉnh đã trình bày trên bởi hàm đệ quy Solvable(u). Hàm này nhận giá trị true nếu u giải được và nhận giá trị ***false*** nếu u không giải được. Trong hàm Solvable(u), ta sẽ sử dụng:

- Biến Ok. Với mỗi toán tử R áp dụng được tại u, biến Ok nhận giá trị ***true*** nếu tất cả các đỉnh v kề u theo R đều giải được, và Ok nhận giá trị ***false*** nếu có một đỉnh v kề u theo R không giải được.

- Hàm Operator(u) ghi lại toán tử áp dụng thành công tại u, tức là Operator(u) = R nếu mọi đỉnh v kề u theo R đều giải được.

function *Solvable(u)***;**

begin

1. **if** *u là đỉnh kết thúc* **then**

{*Solvable* ← **true; stop};**

2. **if** *u không là đỉnh kết thúc và không có đỉnh kề* **then**

{*Solvable(u)* ← **false; stop};**

3. **for** *mỗi toán tử R áp dụng được tại u* **do**

{*Ok* ← *true***;**

for *mỗi v kề u theo R* **do**

if *Solvable(v)* = **false then {***Ok* ← **false; exit};**

if *Ok* **then**

{*Solvable(u)* ← **true;** *Operator(u)* ← *R***; stop}}**

4. Solvable(u) ← **false;**

end;

Nhận xét

 Hoàn toàn tương tự như thuật toán tìm kiếm theo độ sâu trong không gian trạng thái (mục 1.3.2), thuật toán tìm kiếm theo độ sâu trên đồ thị và/hoặc sẽ xác định được bài toán ban đầu là giải được hay không giải được, nếu cây tìm kiếm không có nhánh vô hạn. Nếu cây tìm kiếm có nhánh vô hạn thì chưa chắc thuật toán đã dừng, vì có thể nó bị xa lầy khi đi xuống nhánh vô hạn. Trong trường hợp này ta nên sử dụng thuật toán tìm kiếm sâu lặp (mục 1.3.3).

Nếu bài toán ban đầu giải được, thì bằng cách sử dụng hàm Operator ta sẽ xây dựng được cây nghiệm.

Chương II

Các chiến lược tìm kiếm kinh nghiệm

Trong chương I, chúng ta đã nghiên cứu việc biểu diễn vấn đề trong không gian trạng thái và các kỹ thuật tìm kiếm mù. Các kỹ thuật tìm kiếm mù rất kém hiệu quả và trong nhiều trường hợp không thể áp dụng được. Trong chương này, chúng ta sẽ nghiên cứu các phương pháp tìm kiếm kinh nghiệm (tìm kiếm heuristic), đó là các phương pháp sử dụng hàm đánh giá để hướng dẫn sự tìm kiếm.

Hàm đánh giá và tìm kiếm kinh nghiệm:

Trong nhiều vấn đề, ta có thể sử dụng kinh nghiệm, tri thức của chúng ta về vấn đề để đánh giá các trạng thái của vấn đề. Với mỗi trạng thái u, chúng ta sẽ xác định một giá trị số h(u), số này đánh giá "sự gần đích" của trạng thái u. Hàm h(u) được gọi là ***hàm đánh giá***. Chúng ta sẽ sử dụng hàm đánh giá để hướng dẫn sự tìm kiếm. Trong quá trình tìm kiếm, tại mỗi bước ta sẽ chọn trạng thái để phát triển là trạng thái có giá trị hàm đánh giá nhỏ nhất, trạng thái này được xem là trạng thái có nhiều hứa hẹn nhất hướng tới đích.

Các kỹ thuật tìm kiếm sử dụng hàm đánh giá để hướng dẫn sự tìm kiếm được gọi chung là các kỹ thuật tìm kiếm kinh nghiệm (heuristic search). Các giai đoạn cơ bản để giải quyết vấn đề bằng tìm kiếm kinh nghiệm như sau:

1. Tìm biểu diễn thích hợp mô tả các trạng thái và các toán tử của vấn đề.

2. Xây dựng hàm đánh giá.

3. Thiết kế chiến lược chọn trạng thái để phát triển ở mỗi bước.

Hàm đánh giá

Trong tìm kiếm kinh nghiệm, hàm đánh giá đóng vai trò cực kỳ quan trọng. Chúng ta có xây dựng được hàm đánh giá cho ta sự đánh giá đúng các trạng thái thì tìm kiếm mới hiệu quả. Nếu hàm đánh giá không chính xác, nó có thể dẫn ta đi chệch hướng và do đó tìm kiếm kém hiệu quả.

Hàm đánh giá được xây dựng tùy thuộc vào vấn đề. Sau đây là một số ví dụ về hàm đánh giá:

- Trong bài toán tìm kiếm đường đi trên bản đồ giao thông, ta có thể lấy độ dài của đường chim bay từ một thành phố tới một thành phố đích làm giá trị của hàm đánh giá.

- Bài toán 8 số. Chúng ta có thể đưa ra hai cách xây dựng hàm đánh giá.

Hàm h_1: Với mỗi trạng thái u thì $h_1(u)$ là số quân không nằm đúng vị trí của nó trong trạng thái đích. Chẳng hạn trạng thái đích ở bên phải hình 2.1, và u là trạng thái ở bên trái hình 2.1, thì $h_1(u) = 4$, vì các quân không đúng vị trí là 3, 8, 6 và 1.

u =

3	2	8
	6	4
7	1	5

1	2	3
8		4
7	6	5

$h_1(u) = 4$ $h_2(u) = 9$

Hình 2.1 Đánh giá trạng thái u.

Hàm h_2: $h_2(u)$ là tổng khoảng cách giữa vị trí của các quân trong trạng thái u và vị trí của nó trong trạng thái đích. ở đây khoảng cách được hiểu là số ít nhất các dịch chuyển theo hàng hoặc cột để đưa một quân tới vị trí của nó trong trạng thái đích. Chẳng hạn với trạng thái u và trạng thái đích như trong hình 2.1, ta có:

$$\mathbf{h_2(u) = 2 + 3 + 1 + 3 = 9}$$

Vì quân 3 cần ít nhất 2 dịch chuyển, quân 8 cần ít nhất 3 dịch chuyển, quân 6 cần ít nhất 1 dịch chuyển và quân 1 cần ít nhất 3 dịch chuyển.

Hai chiến lược tìm kiếm kinh nghiệm quan trọng nhất là tìm kiếm tốt nhất - đầu tiên (best-first search) và tìm kiếm leo đồi (hill-climbing search). Có thể xác định các chiến lược này như sau:

Tìm kiếm tốt nhất đầu tiên = Tìm kiếm theo bề rộng + Hàm đánh giá

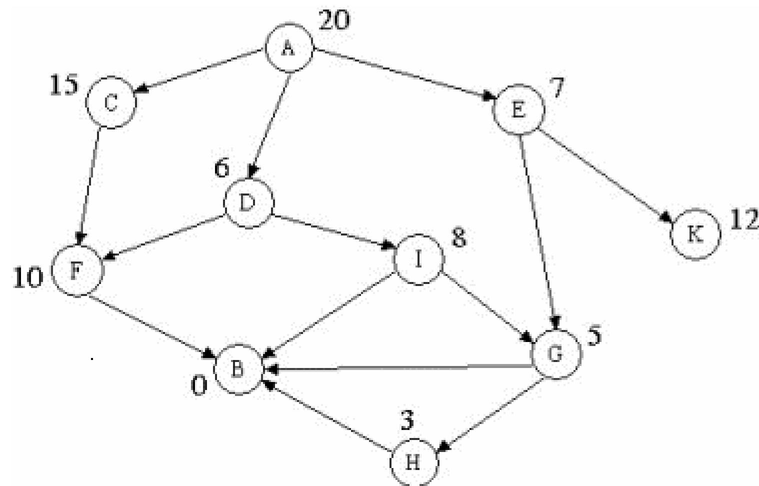
Tìm kiếm leo đồi = Tìm kiếm theo độ sâu + Hàm đánh giá

Chúng ta sẽ lần lượt nghiên cứu các kỹ thuật tìm kiếm này trong các mục sau.

Tìm kiếm tốt nhất - đầu tiên:

Tìm kiếm tốt nhất - đầu tiên (best-first search) là tìm kiếm theo bề rộng được hướng dẫn bởi hàm đánh giá. Nhưng nó khác với tìm kiếm theo bề rộng ở chỗ, trong tìm kiếm theo bề rộng ta lần lượt phát triển tất cả các đỉnh ở mức hiện tại để sinh ra các đỉnh ở mức tiếp theo, còn trong tìm kiếm tốt nhất - đầu tiên ta chọn

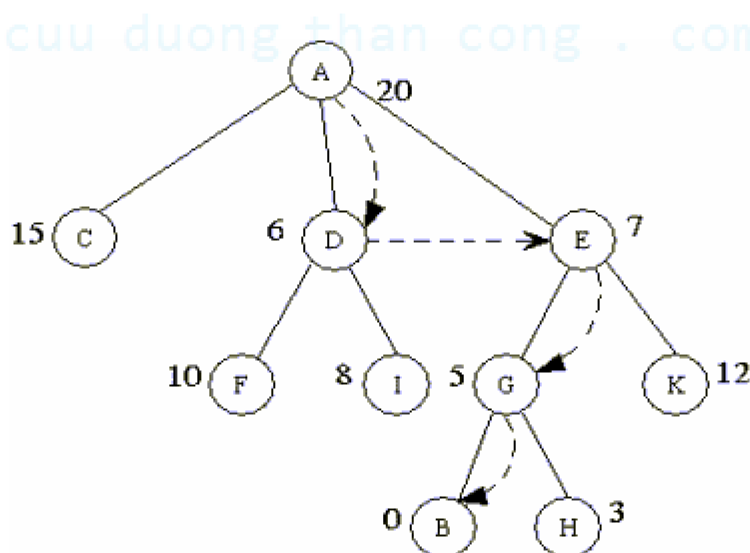
đỉnh để phát triển là đỉnh tốt nhất được xác định bởi hàm đánh giá (tức là đỉnh có giá trị hàm đánh giá là nhỏ nhất), đỉnh này có thể ở mức hiện tại hoặc ở các mức



Hình 2.2 Đồ thị không gian trạng thái

trên.

Ví dụ: Xét không gian trạng thái được biểu diễn bởi đồ thị trong hình 2.2, trong đó trạng thái ban đầu là A, trạng thái kết thúc là B. Giá trị của hàm đánh giá là các số ghi cạnh mỗi đỉnh. Quá trình tìm kiếm tốt nhất - đầu tiên diễn ra như sau: Đầu tiên phát triển đỉnh A sinh ra các đỉnh kề là C, D và E. Trong ba đỉnh này, đỉnh D có giá trị hàm đánh giá nhỏ nhất, nó được chọn để phát triển và sinh ra F, I. Trong số các đỉnh chưa được phát triển C, E, F, I thì đỉnh E có giá trị đánh giá nhỏ nhất, nó được chọn để phát triển và sinh ra các đỉnh G, K. Trong số các đỉnh chưa được phát triển thì G tốt nhất, phát triển G sinh ra B, H. Đến đây ta đã đạt tới trạng thái kết thúc. Cây tìm kiếm tốt nhất - đầu tiên được biểu diễn trong hình 2.3.



Hình 2.3 Cây tìm kiếm tốt nhất - đầu tiên

Sau đây là thủ tục tìm kiếm tốt nhất - đầu tiên. Trong thủ tục này, chúng ta sử dụng danh sách L để lưu các trạng thái chờ phát triển, danh sách được sắp theo thứ tự tăng dần của hàm đánh giá sao cho trạng thái có giá trị hàm đánh giá nhỏ nhất ở đầu danh sách.

procedure *Best_First_Search*;

begin

1. Khởi tạo danh sách L chỉ chứa trạng thái ban đầu;

2. **loop do**

2.1 **if** *L rỗng* **then**

{*thông báo thất bại*; **stop**};

2.2 Loại trạng thái u ở đầu danh sách L;

2.3 **if** *u là trạng thái kết thúc* **then**

{*thông báo thành công*; **stop**}

2.4 **for** *mỗi trạng thái v kề u* **do**

Xen v vào danh sách L sao cho L được sắp theo thứ tự tăng dần của hàm đánh giá;

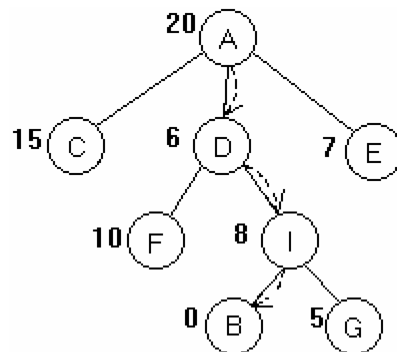
end;

Tìm kiếm leo đồi:

Tìm kiếm leo đồi (hill-climbing search) là tìm kiếm theo độ sâu được hướng dẫn bởi hàm đánh giá. Song khác với tìm kiếm theo độ sâu, khi ta phát triển một

đỉnh u thì bước tiếp theo, ta chọn trong số các đỉnh con của u, đỉnh có nhiều hứa hẹn nhất để phát triển, đỉnh này được xác định bởi hàm đánh giá.

Ví dụ: Ta lại xét đồ thị không gian trạng thái trong hình 2.2. Quá trình tìm kiếm leo đồi được tiến hành như sau. Đầu tiên phát triển đỉnh A sinh ra các đỉnh con C, D, E. Trong các đỉnh này chọn D để phát triển, và nó sinh ra các đỉnh con



Hình 2.4 Cây tìm kiếm leo đồi

B, G. Quá trình tìm kiếm kết thúc. Cây tìm kiếm leo đồi được cho trong hình 2.4.

Trong thủ tục tìm kiếm leo đồi được trình bày dưới đây, ngoài danh sách L lưu các trạng thái chờ được phát triển, chúng ta sử dụng danh sách L_1 để lưu giữ tạm thời các trạng thái kề trạng thái u, khi ta phát triển u. Danh sách L_1 được sắp xếp theo thứ tự tăng dần của hàm đánh giá, rồi được chuyển vào danh sách L sao trạng thái tốt nhất kề u đứng ở danh sách L.

procedure *Hill_Climbing_Search*;

begin

1. Khởi tạo danh sách L chỉ chứa trạng thái ban đầu;

2. **loop do**

2.1 **if** *L rỗng* **then**

{*thông báo thất bại*; **stop**};

2.2 Loại trạng thái u ở đầu danh sách L;

2.3 **if** *u là trạng thái kết thúc* **then**

{*thông báo thành công*; **stop**};

2.3 **for** *mỗi trạng thái v kề u do đặt v vào* L_1;

2.5 Sắp xếp L_1 *theo thứ tự tăng dần của hàm đánh giá*;

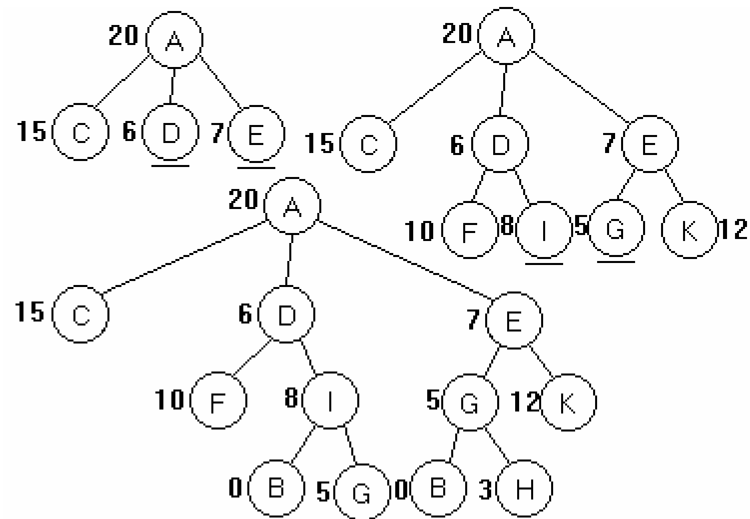
2.6 Chuyển danh sách L_1 *vào đầu danh sách L*;

end;

Tìm kiếm beam

Tìm kiếm beam (beam search) giống như tìm kiếm theo bề rộng, nó phát triển các đỉnh ở một mức rồi phát triển các đỉnh ở mức tiếp theo. Tuy nhiên, trong tìm kiếm theo bề rộng, ta phát triển tất cả các đỉnh ở một mức, còn trong tìm kiếm beam, ta hạn chế chỉ phát triển k đỉnh tốt nhất (các đỉnh này được xác định bởi hàm đánh giá). Do đó trong tìm kiếm beam, ở bất kỳ mức nào cũng chỉ có nhiều nhất k đỉnh được phát triển, trong khi tìm kiếm theo bề rộng, số đỉnh cần phát triển ở mức d là b^d (b là nhân tố nhánh).

Ví dụ: Chúng ta lại xét đồ thị không gian trạng thái trong hình 2.2. Chọn k = 2. Khi đó cây tìm kiếm beam được cho như hình 2.5. Các đỉnh được gạch dưới là



Hình 2.5 Cây tìm kiếm beam.

các đỉnh được chọn để phát triển ở mỗi mức.

Chương III

Các chiến lược tìm kiếm tối ưu

Vấn đề tìm kiếm tối ưu, một cách tổng quát, có thể phát biểu như sau. Mỗi đối tượng x trong không gian tìm kiếm được gắn với một số đo giá trị của đối tượng đó f(x), mục tiêu của ta là tìm đối tượng có giá trị f(x) lớn nhất (hoặc nhỏ nhất) trong không gian tìm kiếm. Hàm f(x) được gọi là hàm mục tiêu. Trong chương này chúng ta sẽ nghiên cứu các thuật toán tìm kiếm sau:

- Các kỹ thuật tìm đường đi ngắn nhất trong không gian trạng thái: Thuật toán A*, thuật toán nhánh_và_cận.

- Các kỹ thuật tìm kiếm đối tượng tốt nhất: Tìm kiếm leo đồi, tìm kiếm gradient, tìm kiếm mô phỏng luyện kim.

- Tìm kiếm bắt chước sự tiến hóa: thuật toán di truyền.

1.1 Tìm đường đi ngắn nhất.

Trong các chương trước chúng ta đã nghiên cứu vấn đề tìm kiếm đường đi từ trạng thái ban đầu tới trạng thái kết thúc trong không gian trạng thái. Trong mục này, ta giả sử rằng, giá phải trả để đưa trạng thái a tới trạng thái b (bởi một toán tử nào đó) là một số k(a,b) ≥ 0, ta sẽ gọi số này là độ dài cung (a,b) hoặc giá trị của cung (a,b) trong đồ thị không gian trạng thái. Độ dài của các cung được xác định tùy thuộc vào vấn đề. Chẳng hạn, trong bài toán tìm đường đi trong bản đồ giao thông, giá của cung (a,b) chính là độ dài của đường nối thành phố a với thành phố

b. Độ dài đường đí được xác định là tổng độ dài của các cung trên đường đi. Vấn đề của chúng ta trong mục này, tìm đường đi ngắn nhất từ trạng thái ban đầu tới trạng thái đích. Không gian tìm kiếm ở đây bao gồm tất cả các đường đi từ trạng thái ban đầu tới trạng thái kết thúc, hàm mục tiêu được xác định ở đây là độ dài của đường đi.

Chúng ta có thể giải quyết vấn đề đặt ra bằng cách tìm tất cả các đường đi có thể có từ trạng thái ban đầu tới trạng thái đích (chẳng hạn, sử sụng các ký thuật tìm kiếm mù), sau đó so sánh độ dài của chúng, ta sẽ tìm ra đường đi ngắn nhất. Thủ tục tìm kiếm này thường được gọi là thủ tục bảo tàng Anh Quốc (British Museum Procedure). Trong thực tế, kỹ thuật này không thể áp dụng được, vì cây tìm kiếm thường rất lớn, việc tìm ra tất cả các đường đi có thể có đòi hỏi rất nhiều thời gian. Do đó chỉ có một cách tăng hiệu quả tìm kiếm là sử dụng các hàm đánh giá đề hướng dẫn sử tìm kiếm. Các phương pháp tìm kiếm đường đi ngắn nhất mà chúng ta sẽ trình bày đều là các phương pháp tìm kiếm heuristic.

Giả sử u là một ***trạng thái đạt tới*** (có dường đi từ trạng thái ban đầu u_0 tới u). Ta xác định hai hàm đánh giá sau:

- g(u) là đánh giá độ dài đường đi ngắn nhất từ u_0 tới u (Đường đi từ u_0 tới trạng thái u không phải là trạng thái đích được gọi là ***đường đi một phần***, để phân biệt với ***đường đi đầy đủ***, là đường đi từ u_0 tới trạng thái đích).

- h(u) là đánh giá độ dài đường đi ngắn nhất từ u tới trạng thái đích.

Hàm h(u) được gọi là ***chấp nhận được*** (hoặc đánh giá thấp***)*** nếu với mọi trạng thái u, h(u) ≤ độ dài đường đi ngắn nhất thực tế từ u tới trạng thái đích. Chẳng hạn trong bài toán tìm đường đi ngắn nhất trên bản đồ giao thông, ta có thể xác định h(u) là độ dài đường chim bay từ u tới đích.

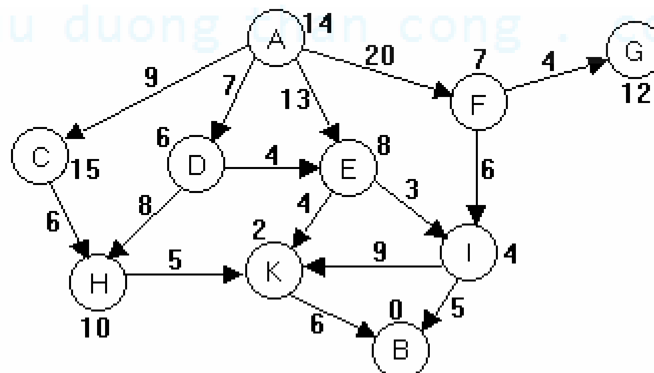
Ta có thể sử dụng kỹ thuật tìm kiếm leo đồi với hàm đánh giá h(u). Tất nhiên phương pháp này chỉ cho phép ta tìm được đường đi tương đối tốt, chưa chắc đã là đường đi tối ưu.

Ta cũng có thể sử dụng kỹ thuật tìm kiếm tốt nhất đầu tiên với hàm đánh giá g(u). Phương pháp này sẽ tìm ra đường đi ngắn nhất, tuy nhiên nó có thể kém hiệu quả.

Để tăng hiệu quả tìm kiếm, ta sử dụng hàm đánh giá mới :

f(u) = g(u) + h(u)

Tức là, f(u) là đánh giá độ dài đường đi ngắn nhất qua u từ trạng thái ban đầu tới trạng thái kết thúc.



Hình 3.1 Đồ thị không gian trạng thái với hàm đánh giá.

*1.1.1 Thuật toán A**

Thuật toán A* là thuật toán sử dụng kỹ thuật tìm kiếm tốt nhất đầu tiên với hàm đánh giá f(u).

Để thấy được thuật toán A* làm việc như thế nào, ta xét đồ thị không gian trạng thái trong hình 3.1. Trong đó, trạng thái ban đầu là trạng thái A, trạng thái đích là B, các số ghi cạnh các cung là độ dài đường đi, các số cạnh các đỉnh là giá

trị của hàm h.Đầu tiên, phát triển đỉnh A sinh ra các đỉnh con C, D, E và F. Tính giá trị của hàm f tại các đỉnh này ta có:

g(C) = 9, f(C) = 9 + 15 = 24, g(D) = 7, f(D) = 7 + 6 = 13,

g(E) = 13, f(E) = 13 + 8 = 21, g(F) = 20, f(F) = 20 +7 = 27

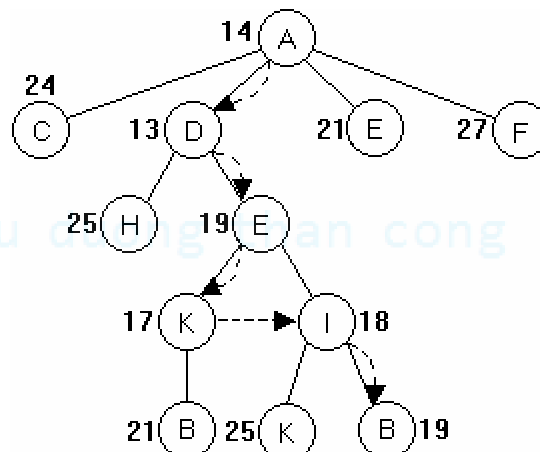
Như vậy đỉnh tốt nhất là D (vì f(D) = 13 là nhỏ nhất). Phát triển D, ta nhận được các đỉnh con H và E. Ta đánh giá H và E (mới):

g(H) = g(D) + Độ dài cung (D, H) = 7 + 8 = 15, f(H) = 15 + 10 = 25.

Đường đi tới E qua D có độ dài:

g(E) = g(D) + Độ dài cung (D, E) = 7 + 4 = 11.

Vậy đỉnh E mới có đánh giá là f(E) = g(E) + h(E) = 11 + 8 = 19. Trong số các đỉnh cho phát triển, thì đỉnh E với đánh giá f(E) = 19 là đỉnh tốt nhất. Phát triển đỉnh này, ta nhận được các đỉnh con của nó là K và I. Chúng ta tiếp tục quá trình trên cho tới khi đỉnh được chọn để phát triển là đỉnh kết thúc B, độ dài đường đi ngắn nhất tới B là g(B) = 19. Quá trình tìm kiếm trên được mô tả bởi cây tìm



Hình 3.2 Cây tìm kiếm theo thuật toán A*

kiếm trong hình 3.2, trong đó các số cạnh các đỉnh là các giá trị của hàm đánh giá f(u).

procedure *A**;

begin

1. Khởi tạo danh sách L chỉ chứa trạng thái ban đầu;

2. **loop do**

2.1 **if** *L rỗng* **then**

{*thông báo thất bại*; **stop};**

2.2 Loại trạng thái u ở đầu danh sách L;

2.3 **if** *u là trạng thái đích* **then**

{*thông báo thành công*; **stop}**

2.4 **for** *mỗi trạng thái v kề u* **do**

{$g(v) \leftarrow g(u) + k(u,v)$**;**

$f(v) \leftarrow g(v) + h(v)$**;**

Đặt v vào danh sách L;**}**

2.5 Sắp xếp L theo thứ tự tăng dần của hàm f sao cho

trạng thái có giá trị của hàm f nhỏ nhất

ở đầu danh sách;

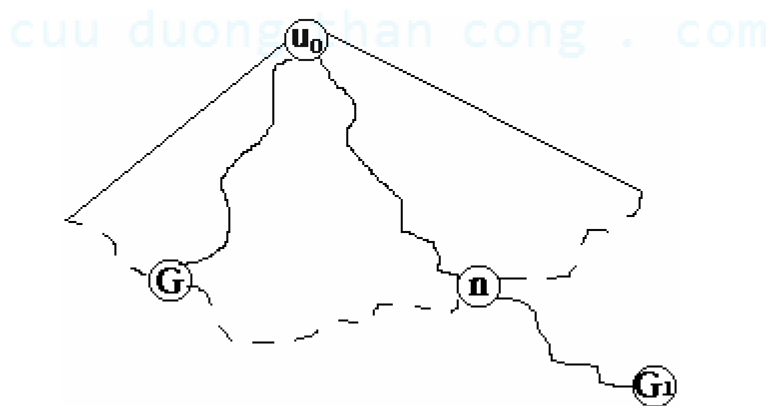
end;

Chúng ta đưa ra một số nhận xét về thuật toán A*.

- Người ta chứng minh được rằng, nếu hàm đánh giá h(u) là đánh giá thấp nhất (trường hợp đặc biệt, h(u) = 0 với mọi trạng thái u) thì thuật toán A* là thuật toán ***tối ưu,*** tức là nghiệm mà nó tìm ra là nghiệm tối ưu. Ngoài ra, nếu độ dài của các cung không nhỏ hơn một số dương δ nào đó thì thuật toán A* là thuật toán ***đầy đủ*** theo nghĩa rằng, nó luôn dừng và tìm ra nghiệm.

Chúng ta chứng minh tính tối ưu của thuật toán A*.

Giả sử thuật toán dừng lại ở đỉnh kết thúc G với độ dài đường đi từ trạng thái ban đầu u_0 tới G là g(G). Vì G là đỉnh kết thúc, ta có h(G) = 0 và f(G) = g(G) +



Hình 3.3 Đỉnh lá n của cây tìm kiếm nằm trên đường đi tối ưu.

h(G) = g(G). Giả sử nghiệm tối ưu là đường đi từ u_0 tới đỉnh kết thúc G_1 với độ dài l. Giả sử đường đi này "thoát ra" khỏi cây tìm kiếm tại đỉnh lá n (Xem hình 3.3). Có thể xẩy ra hai khả năng: n trùng với G_1 hoặc không. Nếu n là G_1 thì vì G được chọn để phát triển trước G_1, nên $f(G) \le f(G_1)$, do đó $g(G) \le g(G_1) = l$. Nếu $n \ne G_1$ thì do h(u) là hàm đánh giá thấp, nên $f(n) = g(n) + h(n) \le l$. Mặt khác, cũng do G được chọn để phát triển trước n, nên $f(G) \le f(n)$, do đó, $g(G) \le l$. Như vậy, ta đã

chứng minh được rằng độ dài của đường đi mà thuật toán tìm ra g(G) không dài hơn độ dài l của đường đi tối ưu. Vậy nó là độ dài đường đi tối ưu.

- Trong trường hợp hàm đánh giá h(u) = 0 với mọi u, thuật toán A* chính là thuật toán tìm kiếm tốt nhất đầu tiên với hàm đánh giá g(u) mà ta đã nói đến.

- Thuật toán A* đã được chứng tỏ là thuật toán hiệu quả nhất trong số các thuật toán đầy đủ và tối ưu cho vấn đề tìm kiếm đường đi ngắn nhất.

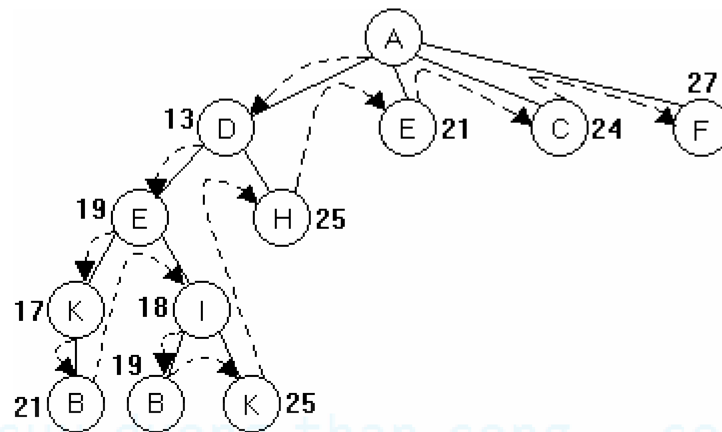
1.1.2 Thuật toán tìm kiếm nhánh-và-cận.

Thuật toán nhánh_và_cận là thuật toán sử dụng tìm kiếm leo đồi với hàm đánh giá f(u).

Trong thuật toán này, tại mỗi bước khi phát triển trạng thái u, thì ta sẽ chọn trạng thái tốt nhất v (f(v) nhỏ nhất) trong số các trạng thái kề u đề phát triển ở bước sau. Đi xuống cho tới khi gặp trạng thái v là đích, hoặc gặp trạng thái v không có đỉnh kề, hoặc gặp trạng thái v mà f(v) lớn hơn độ dài đường đi tối ưu tạm thời, tức là đường đi đầy đủ ngắn nhất trong số các đường đi đầy đủ mà ta đã tìm ra. Trong các trường hợp này, ta không phát triển đỉnh v nữa, hay nói cách khác, ta cất đi các nhánh cây xuất phát từ v, và quay lên cha của v đề tiếp tục đi xuống trạng thái tốt nhất trong các trạng thái còn lại chưa được phát triển.

Ví dụ: Chúng ta lại xét không gian trạng thái trong hình 3.1. Phát triển đỉnh A, ta nhận được các đỉnh con C, D, E và F, f(C) = 24, f(D) = 13, f(E) = 21, f(F) = 27. Trong số này D là tốt nhất, phát triển D, sinh ra các đỉnh con H và E, f(H) = 25, f(E) = 19. Đi xuống phát triển E, sinh ra các đỉnh con là K và I, f(K) = 17, f(I) = 18. Đi xuống phát triển K sinh ra đỉnh B với f(B) = g(B) = 21. Đi xuống B, vì B là đỉnh đích, vậy ta tìm được đường đi tối ưu tạm thời với độ dài 21. Từ B quay lên K, rồi từ K quay lên cha nó là E. Từ E đi xuống J, f(J) = 18 nhỏ hơn độ dài đường đi tạm thời (là 21). Phát triển I sinh ra các con K và B, f(K) = 25, f(B) =

g(B) = 19. Đi xuống đỉnh B, vì đỉnh B là đích ta tìm được đường đi đầy đủ mới với độ dài là 19 nhỏ hơn độ dài đường đi tối ưu tạm thời cũ (21). Vậy độ dài đường đi tối ưu tạm thời bây giờ là 19. Bây giờ từ B ta lại quay lên các đỉnh còn lại chưa được phát triển. Song các đỉnh này đều có giá trị hàm đánh giá lớn hơn 19, do đó không có đỉnh nào được phát triển nữa. Như vậy, ta tìm được đường đi tối ưu với độ dài 19. Cây tìm kiếm được biểu diễn trong hình 3.4.



Hình 3.4 Cây tìm kiếm nhánh_và_cận.

Thuật toán nhánh_và_cận sẽ được biểu diễn bởi thủ tục Branch_and_Bound. Trong thủ tục này, biến cost được dùng để lưu độ dài đường đi ngắn nhất. Giá trị ban đầu của cost là số đủ lớn, hoặc độ dài của một đường đi đầy đủ mà ta đã biết.

procedure *Branch_and_Bound***;**

begin

*1. Khởi tạo danh sách L chỉ chứa trạng thái ban đầu***;**

*Gán giá trị ban đầu cho cost***;**

2. **loop do**

2.1 **if** *L rỗng* **then stop;**

2.2 Loại trạng thái u ở đầu danh sách L;

2.3 **if** *u là trạng thái kết thúc* **then**

if $g(u) \le y$ **then** {$y \leftarrow g(y)$; *Quay lại 2.1*};

2.4 **if** $f(u) > y$ **then** *Quay lại 2.1*;

2.5 **for** *mỗi trạng thái v kề u* **do**

{$g(v) \leftarrow g(u) + k(u,v)$;

$f(v) \leftarrow g(v) + h(v)$;

Đặt v vào danh sách L_1};

2.6 Sắp xếp L_1 *theo thứ tự tăng của hàm f*;

2.7 Chuyển L_1 *vào đầu danh sách L sao cho trạng thái*

ở đầu L_1 *trở thành ở đầu L*;

end;

Người ta chứng minh được rằng, thuật toán nhánh_và_cận cũng là thuật toán đầy đủ và tối ưu nếu hàm đánh giá h(u) là đánh giá thấp và có độ dài các cung không nhỏ hơn một số dương δ nào đó.

1.2 Tìm đối tượng tốt nhất

Trong mục này chúng ta sẽ xét vấn đề tìm kiếm sau. Trên không gian tìm kiếm U được xác định hàm giá (hàm mục tiêu) cost, ứng với mỗi đối tượng x ∈ U với một giá trị số cost(x), số này được gọi là giá trị của x. Chúng ta cần tìm một

đối tượng mà tại đó hàm giá trị lớn nhất, ta gọi đối tượng đó là ***đối tượng tốt nhất***. Giả sử không gian tìm kiếm có cấu trúc cho phép ta xác định được khái niệm lân cận của mỗi đối tượng. Chẳng hạn, U là không gian trạng thái thì lân cận của trạng thái u gồm tất cả các trạng thái v kề u; nếu U là không gian các vectơ thực n-chiều thì lân cận của vectơ x = (x_1, x_2, ... x_n) gồm tất cả các vectơ ở gần x theo khoảng cách Ơcơlit thông thường.

Trong mục này, ta sẽ xét kỹ thuật tìm kiếm leo đồi để tìm đối tượng tốt nhất. Sau đó ta sẽ xét kỹ thuật tìm kiếm gradient (gradient search). Đó là kỹ thuật leo đồi áp dụng cho không gian tìm kiếm là không gian các vectơ thực n-chiều và hàm giá là là hàm khả vi liên tục. Cuối cùng ta sẽ nghiên cứu kỹ thuật tìm kiếm mô phỏng luyện kim(simulated annealing).

1.2.1 Tìm kiếm leo đồi

Kỹ thuật tìm kiếm leo đồi để tìm kiếm đối tượng tốt nhất hoàn toàn giống như kỹ thuật tìm kiếm leo đồi để tìm trạng thái kết thúc đã xét trong mục 2.3. Chỉ khác là trong thuật toán leo đồi ở mục 2.3, từ một trạng thái ta "leo lên" trạng thái kề tốt nhất (được xác định bởi hàm giá), tiếp tục cho tới khi đạt tới trạng thái đích; nếu chưa đạt tới trạng thái đích mà không leo lên được nữa, thì ta tiếp tục "tụt xuống" trạng thái trước nó, rồi lại leo lên trạng thái tốt nhất còn lại. Còn ở đây, từ một đỉnh u ta chỉ leo lên đỉnh tốt nhất v (được xác định bởi hàm giá cost) trong lân cận u nếu đỉnh này "cao hơn" đỉnh u, tức là cost(v) > cost(u). Quá trình tìm kiếm sẽ dừng lại ngay khi ta không leo lên đỉnh cao hơn được nữa.

Trong thủ tục leo đồi dưới đây, biến u lưu đỉnh hiện thời, biến v lưu đỉnh tốt nhất (cost(v) nhỏ nhất) trong các đỉnh ở lân cận u. Khi thuật toán dừng, biến u sẽ lưu trong đối tượng tìm được.

procedure *Hill_Climbing*;

begin

*1. u ← một đối tượng ban đầu nào đó***;**

2. **if** *cost(v) > cost(u)* **then** *u ← v* **else stop;**

end;

Tối ưu địa phương và tối ưu toàn cục

Rõ ràng là, khi thuật toán leo đồi dừng lại tại đối tương u*, thì giá của nó cost(u*) lớn hơn giá của tất cả các đối tượng nằm trong lân cận của tất cả các đối tượng trên đường đi từ đối tượng ban đầu tới trạng thái u*. Do đó nghiệm u* mà thuật toán leo đồi tìm được là ***tối ưu địa phương***. Cần nhấn mạnh rằng không có gì đảm bảo nghiệm đó là ***tối ưu toàn cục*** theo nghĩa là cost(u*) là lớn nhất trên toàn bộ không gian tìm kiếm.

Để nhận được nghiệm tốt hơn bằng thuật toán leo đồi, ta có thể áp dụng lặp lại nhiều lần thủ tục leo đồi xuất phát từ một dãy các đối tượng ban đầu được chọn ngẫu nhiên và lưu lại nghiệm tốt nhất qua mỗi lần lặp. Nếu số lần lặp đủ lớn thì ta có thể tìm được nghiệm tối ưu.

Kết quả của thuật toán leo đồi phụ thuộc rất nhiều vào hình dáng của "mặt cong" của hàm giá. Nếu mặt cong chỉ có một số ít cực đại địa phương, thì kỹ thuật leo đồi sẽ tìm ra rất nhanh cực đại toàn cục. Song có những vấn đề mà mặt cong của hàm giá tựa như lông nhím vậy, khi đó sử dụng kỹ thuật leo đồi đòi hỏi rất nhiều thời gian.

1.2.2 Tìm kiếm gradient

Tìm kiếm gradient là kỹ thuật tìm kiếm leo đồi để tìm giá trị lớn nhất (hoặc nhỏ nhất) của hàm khả vi liên tục f(x) trong không gian các vectơ thực n-chiều.

$$\nabla f = \left(\frac{\partial f}{\partial x1}, \frac{\partial f}{\partial x2}, \ldots, \frac{\partial f}{\partial xn} \right)$$

Như ta đã biết, trong lân cận đủ nhỏ của điểm x = (x_1,...,x_n), thì hàm f tăng nhanh nhất theo hướng của vectơ gradient:

Do đó tư tưởng của tìm kiếm gradient là từ một điểm ta đi tới điểm ở lân cận nó theo hướng của vectơ gradient.

procedure *Gradient_Search*;

begin

x ← điểm xuất phát nào đó;

repeat

$x \leftarrow x + \alpha \nabla f(x)$;

until $|\nabla f| < \varepsilon$;

end;

Trong thủ tục trên, α là hằng số dương nhỏ nhất xác định tỉ lệ của các bước, còn ε là hằng số dương nhỏ xác định tiêu chuẩn dừng. Bằng cách lấy các bước đủ nhỏ theo hướng của vectơ gradient chúng ta sẽ tìm được điểm cực đại địa phương, đó là điểm mà tại đó ∇f = 0, hoặc tìm được điểm rất gần vói cực đại địa phương.

1.2.3 Tìm kiếm mô phỏng luyện kim:

Như đã nhấn mạnh ở trên, tìm kiếm leo đồi không đảm bảo cho ta tìm được nghiệm tối ưu toàn cục. Để cho nghiệm tìm được gần với tối ưu toàn cục, ta áp dụng kỹ thuật leo đồi lặp xuất phát từ các điểm được lựa chọn ngẫu nhiên. Bây giờ thay cho việc luôn luôn "leo lên đồi" xuất phát từ các điểm khác nhau, ta thực hiện

một số bước "tụt xuống" nhằm thoát ra khỏi các điểm cực đại địa phương. Đó chính là tư tưởng của kỹ thuật tìm kiếm mô phỏng luyện kim.

Trong tìm kiếm leo đồi, khi ở một trạng thái u ta luôn luôn đi tới trạng thái tốt nhất trong lân cận nó. Còn bây giờ, trong tìm kiếm mô phỏng luyện kim, ta chọn ngẫu nhiên một trạng thái v trong lân cận u. Nếu trạng thái v được chọn tốt hơn u (cost(v) > cost(u)) thì ta đi tới v, còn nếu không ta chỉ đi tới v với một xác suất nào đó. Xác suất này giảm theo hàm mũ của "độ xấu" của trạng thái v. Xác suất này còn phụ thuộc vào tham số nhiệt độ T. Nhiệt độ T càng cao thì bước đi tới trạng thái xấu càng có khả năng được thực hiện. Trong quá trình tìm kiếm, tham số nhiệt độ T giảm dần tới không. Khi T gần không, thuật toán hoạt động gần giống như leo đồi, hầu như nó không thực hiện bước tụt xuống. Cụ thể ta xác định xác suất đi tới trạng thái xấu v từ u là $e^{\Delta/T}$, ở đây Δ = cost(v) - cost(u).

Sau đây là thủ tục mô phỏng luyện kim.

procedure *Simulated_Anneaning*;

begin

t ← *0*;

u ← *trạng thái ban đầu nào đó*;

T ← *nhiệt độ ban đầu*;

repeat

v ← *trạng thái được chọn nhẫu nhiên trong lân cận u*;

if *cost(v)* > *cost(u)* **then** *u* ← *v*

else *u* ← *v với xác suất* $e^{\Delta/T}$**;**

T ← *g(T, t)***;**

t ← *t* + *1***;**

until *T đủ nhỏ*

end;

Trong thủ tục trên, hàm g(T, t) thỏa mãn điều kiện g(T, t) < T với mọi t, nó xác định tốc độ giảm của nhiệt độ T. Người ta chứng minh được rằng, nếu nhiêt độ T giảm đủ chậm, thì thuật toán sẽ tìm được nghiệm tối ưu toàn cục. Thuật toán mô phỏng luyện kim đã được áp dụng thành công cho các bài toán tối ưu cỡ lớn.

1.3 Tìm kiếm mô phỏng sự tiến hóa. Thuật toán di truyền

Thuật toán di truyền (TTDT) là thuật toán bắt chước sự chọn lọc tự nhiên và di truyền. Trong tự nhiên, các cá thể khỏe, có khả năng thích nghi tốt với môi trường sẽ được tái sinh và nhân bản ở các thế hệ sau. Mỗi cá thể có cấu trúc gien đặc trưng cho phẩm chất của cá thể đó. Trong quá trình sinh sản, các cá thể con có thể thừa hưởng các phẩm chất của cả cha và mẹ, cấu trúc gien của nó mang một phần cấu trúc gien của cha và mẹ. Ngoài ra, trong quá trình tiến hóa, có thể xảy ra hiện tượng đột biến, cấu trúc gien của cá thể con có thể chứa các gien mà cả cha và mẹ đều không có.

Trong TTDT, mỗi cá thể được mã hóa bởi một cấu trúc dữ liệu mô tả cấu trúc gien của cá thể đó, ta sẽ gọi nó là ***nhiễm sắc thể (chroniosome).*** Mỗi nhiễm sắc thể được tạo thành từ các đơn vị được gọi là gien. Chẳng hạn, trong các TTDT cổ điển, các nhiễm sắc thể là các chuỗi nhị phân, tức là mỗi cá thể được biểu diễn bởi một chuỗi nhị phân.

TTDT sẽ làm việc trên các quần thể gồm nhiều cá thể. Một quần thể ứng với một giai đoạn phát triển sẽ được gọi là một ***thế hệ.*** Từ thế hệ ban đầu được tạo ra, TTDT bắt chước chọn lọc tự nhiên và di truyền để biến đổi các thế hệ. TTDT sử dụng các toán tử cơ bản sau đây để biến đổi các thế hệ.

- ***Toán tử tái sinh (reproduction)*** (còn được gọi là ***toán tử chọn lọc (selection)***). Các cá thể tốt được chọn lọc để đưa vào thế hệ sau. Sự lựa chọn này được thực hiện dựa vào độ thích nghi với môi trường của mỗi cá thể. Ta sẽ gọi hàm ứng mỗi cá thể với độ thích nghi của nó là ***hàm thích nghi (fitness function).***

- ***Toán tử lai ghép (crossover).*** Hai cá thể cha và mẹ trao đổi các gien để tạo ra hai cá thể con.

- ***Toán tử đột biến (mutation).*** Một cá thể thay đổi một số gien để tạo thành cá thể mới.

Tất cả các toán tử trên khi thực hiện đều mang tính ngẫu nhiên. Cấu trúc cơ bản của TTDT là như sau:

procedure *Genetic_Algorithm***;**

begin

*t ← 0***;**

*Khởi tạo thế hệ ban đầu P(t)***;**

*Đánh giá P(t) (theo hàm thích nghi)***;**

repeat

*t ← t + 1***;**

Sinh ra thế hệ mới P(t) từ P(t-1) bởi

- *Chọn lọc*
- *Lai ghép*
- *Đột biến;*

Đánh giá P(t);

until *điều kiện kết thúc được thỏa mãn*;

end;

Trong thủ tục trên, điều kiện kết thúc vòng lặp có thể là một số thế hệ đủ lớn nào đó, hoặc độ thích nghi của các cá thể tốt nhất trong các thế hệ kế tiếp nhau khác nhau không đáng kể. Khi thuật toán dừng, cá thể tốt nhất trong thế hệ cuối cùng được chọn làm nghiệm cần tìm.

Bây giờ ta sẽ xét chi tiết hơn toán tử chọn lọc và các toán tử di truyền (lai ghép, đột biến) trong các TTDT cổ điển.

1. Chọn lọc: Việc chọn lọc các cá thể từ một quần thể dựa trên độ thích nghi của mỗi cá thể. Các cá thể có độ thích nghi cao có nhiều khả năng được chọn. Cần nhấn mạnh rằng, hàm thích nghi chỉ cần là ***một hàm thực dương***, nó có thể không tuyến tính, không liên tục, không khả vi. Quá trình chọn lọc được thực hiện theo kỹ thuật quay bánh xe.

Giả sử thế hệ hiện thời P(t) gồm có n cá thể $\{x_1,..,x_n\}$. Số n được gọi là cỡ của quần thể. Với mỗi cá thể x_i, ta tính độ thích nghi của nó $f(x_i)$. Tính tổng các độ

$$F = \sum_{i=1}^{n} f(xi)$$

thích nghi của tất cả các cá thể trong quần thể:

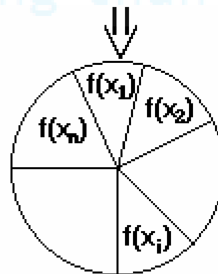
Mỗi lần chọn lọc, ta thực hiện hai bước sau:

- Sinh ra một số thực ngẫu nhiên q trong khoảng (0, F);

$$\sum_{i=1}^{k} f(xi) \geq 4$$

- x_k là cá thể được chọn, nếu k là số nhỏ nhất sao cho

Việc chọn lọc theo hai bước trên có thể minh họa như sau: Ta có một bánh xe được chia thành n phần, mỗi phần ứng với độ thích nghi của một cá thể (hình 3.5). Một mũi tên chỉ vào bánh xe. Quay bánh xe, khi bánh xe dừng, mũi tên chỉ vào phần nào, cá thể ứng với phần đó được chọn.



Hình 3.5 Kỹ thuật quay bánh xe.

Rõ ràng là với cách chọn này, các cá thể có thể có độ thích nghi càng cao càng có khả năng được chọn. Các cá thể có độ thích nghi cao có thể có một hay nhiều bản sao, các cá thể có độ thích nghi thấp có thể không có mặt ở thế hệ sau (nó bị chết đi).

2. Lai ghép: Trên cá thể được chọn lọc, ta tiến hành toán tử lai ghép. Đầu tiên ta cần đưa ra xác suất lai ghép p_c. xác suất này cho ta hy vọng có p_c.n cá thể được lai ghép (n là cỡ của quần thể).

Với mỗi cá thể ta thực hiện hai bước sau:

- Sinh ra số thực ngẫu nhiên r trong đoạn [0, 1];
- Nếu $r < p_c$ thì cá thể đó được chọn để lai ghép

Từ các cá thể được chọn để lai ghép, người ta cặp đôi chúng một cách ngẫu nhiên. Trong trường hợp các nhiễm sắc thể là các chuỗi nhị phân có độ dài cố định m, ta có thể thực hiện lai ghép như sau: Với mỗi cặp, sinh ra một số nguyên ngẫu nhiên p trên đoạn [0, m -1], p là vị trí điểm ghép. Cặp gồm hai nhiễm sắc thể

$$\mathbf{a = (a_1 , ... , a_p , a_{p+1} , ... , a_m)}$$

$$\mathbf{a = (b_1 , ... , b_p , b_{p+1} , ... , b_m)}$$

được thay bởi hai con là:

$$\mathbf{a' = (a_1 , ... , a_p , b_{p+1} , ... , b_m)}$$

$$\mathbf{b' = (b_1 , ... , b_p , a_{p+1} , ... , a_m)}$$

3. Đột biến: Ta thực hiện toán tử đột biến trên các cá thể có được sau quá trình lai ghép. Đột biến là thay đổi trạng thái một số gien nào đó trong nhiễm sắc thể. Mỗi gien chịu đột biến với xác suất p_m. Xác suất đột biến p_m do ta xác định và là xác suất thấp. Sau đây là toán tử đột biến trên các nhiễm sắc thể chuỗi nhị phân.

Với mỗi vị trí i trong nhiễm sắc thể:

$$\mathbf{a = (a_1 , ... , a_i , ... , a_m)}$$

Ta sinh ra một số thực nghiệm ngẫu nhiên p_i trong [0,1]. Qua đột biến a được biến thành a’ như sau:

$$\mathbf{a' = (a'_1 , ... , a'_i , ... , a'_m)}$$

Trong đó :

$$a'_i = \begin{cases} a_i & \text{nếu } p_i \geq p_m \\ 1 - a_i & \text{nếu } p_i < p_m \end{cases}$$

Sau quá trình chọn lọc, lai ghép, đột biến, một thế hệ mới được sinh ra. Công việc còn lại của thuật toán di truyền bây giờ chỉ là lặp lại các bước trên.

Ví dụ: Xét bài toán tìm max của hàm $f(x) = x^2$ với x là số nguyên trên đoạn [0,31]. Để sử dụng TTDT, ta mã hoá mỗi số nguyên x trong đoạn [0,31] bởi một số nhị phân độ dài 5, chẳng hạn, chuỗi 11000 là mã của số nguyên 24. Hàm thích nghi được xác định là chính hàm $f(x) = x^2$. Quần thể ban đầu gồm 4 cá thể (cỡ của quần thể là n = 4). Thực hiện quá trình chọn lọc, ta nhận được kết quả trong bảng sau. Trong bảng này, ta thấy cá thể 2 có độ thích nghi cao nhất (576) nên nó được chọn 2 lần, cá thể 3 có độ thích nghi thấp nhất (64) không được chọn lần nào. Mỗi cá thể 1 và 4 được chọn 1 lần.

Bảng kết quả chọn lọc

Số liệu cá thể	Quần thể ban đầu	x	Độ thích nghi $f(x) = x^2$	Số lần được chọn
1	0 1 1 0 1	13	169	1
2	1 1 0 0 0	24	576	2
3	0 1 0 0 0	8	64	0
4	1 0 0 1 1	19	361	1

Thực hiện qúa trình lai ghép với xác suất lai ghép $p_c = 1$, cả 4 cá thể sau chọn lọc đều được lai ghép. Kết quả lai ghép được cho trong bảng sau. Trong bảng

này, chuỗi thứ nhất được lai ghép với chuỗi thứ hai với điểm ghép là 4, hai chuỗi còn lại được lai ghép với nhau với điểm ghép là 2.

Bảng kết quả lai ghép

Quần thể sau chọn lọc	Điểm ghép	Quần thể sau lai ghép	x	Độ thích nghi f(x) = x2
0 1 1 0 | 1	4	0 1 1 0 0	2	144
1 1 0 0 | 0	4	1 1 0 0 1	5	625
1 1 | 0 0 0	2	1 1 0 1 1	7	729
1 0 | 0 1 1	2	1 0 0 0 0	6	256

Để thực hiện quá trình đột biến, ta chọn xác suất đột biến p_m= 0,001, tức là ta hy vọng có 5.4.0,001 = 0,02 bit được đột biến. Thực tế sẽ không có bit nào được đột biến. Như vậy thế hệ mới là quần thể sau lai ghép. Trong thế hệ ban đầu, độ thích nghi cao nhất là 576, độ thích nghi trung bình 292. Trong thế hệ sau, độ thích nghi cao nhất là 729, trung bình là 438. Chỉ qua một thế hệ, các cá thể đã "tốt lên" rất nhiều.

Thuật toán di truyền khác với các thuật toán tối ưu khác ở các điểm sau:

 TTDT chỉ sử dụng hàm thích để hướng dẫn sự tìm kiếm, hàm thích nghi chỉ cần là hàm thực dương. Ngoài ra, nó không đòi hỏi không gian tìm kiếm phải có cấu trúc nào cả.

- TTDT làm việc trên các nhiễm sắc thể là mã của các cá thể cần tìm.
- TTDT tìm kiếm từ một quần thể gồm nhiều cá thể.
- Các toán tử trong TTDT đều mang tính ngẫu nhiên.

Để giải quyết một vấn đề bằng TTDT, chúng ta cần thực hiện các bước sau đây:

- Trước hết ta cần mã hóa các đối tượng cần tìm bởi một cấu trúc dữ liệu nào đó. Chẳng hạn, trong các TTDT cổ điển, như trong ví dụ trên, ta sử dụng mã nhị phân.
- Thiết kế hàm thích nghi. Trong các bài toán tối ưu, hàm thích nghi được xác định dựa vào hàm mục tiêu.
- Trên cơ sở cấu trúc của nhiễm sắc thể, thiết kế các toán tử di truyền (lai ghép, đột biến) cho phù hợp với các vấn đề cần giải quyết.
- Xác định cỡ của quần thể và khởi tạo quần thể ban đầu.
- Xác định xác suất lai ghép pc và xác suất đột biến. Xác suất đột biến cần là xác suất thấp. Người ta (Goldberg, 1989) khuyên rằng nên chọn xác suất lai ghép là 0,6 và xác suất đột biến là 0,03. Tuy nhiên cần qua thử nghiệm để tìm ra các xác suất thích hợp cho vấn đề cần giải quyết.

Nói chung thuật ngữ TTDT là để chỉ TTDT cổ điển, khi mà cấu trúc của các nhiễm sắc thể là các chuỗi nhị phân với các toán tử di truyền đã được mô tả ở trên. Song trong nhiều vấn đề thực tế, thuận tiện hơn, ta có thể biểu diễn nhiễm sắc thể bởi các cấu trúc khác, chẳng hạn vectơ thực, mảng hai chiều, cây,... Tương ứng với cấu trúc của nhiễm sắc thể, có thể có nhiều cách xác định các toán tử di truyền. Quá trình sinh ra thế hệ mới P(t) từ thế hệ cũ P(t - 1) cũng có nhiều cách chọn lựa.

Người ta gọi chung các thuật toán này là thuật toán tiến hóa (evolutionary algorithms) hoặc chương trình tiến hóa (evolution program).

Thuật toán tiến hóa đã được áp dụng trong các vấn đề tối ưu và học máy. Để hiểu biết sâu sắc hơn về thuật toán tiến hoá, bạn đọc có thể tìm đọc [ ], [ ] và [ ] . [ ] và [ ] được xem là các sách hay nhất viết về TTDT. [ ] cho ta cái nhìn tổng quát về sự phát triển gần đây của TTDT.

Chương IV

Tìm kiếm có đối thủ

Nghiên cứu máy tính chơi cờ đã xuất hiện rất sớm. Không lâu sau khi máy tính lập trình được ra đời vào năm 1950, Claude Shannon đã viết chương trình chơi cờ đầu tiên. các nhà nghiên cứu **Trí Tuệ Nhân Tạo** đã nghiên cứu việc chơi cờ, vì rằng máy tính chơi cờ là một bằng chứng rõ ràng về khả năng máy tính có thể làm được các công việc đòi hỏi trí thông minh của con người. Trong chương này chúng ta sẽ xét các vấn đề sau đây:

- Chơi cờ có thể xem như vấn đề tìm kiếm trong không gian trạng thái.
- Chiến lược tìm kiếm nước đi Minimax.
- Phương pháp cắt cụt α-β, một kỹ thuật để tăng hiệu quả của tìm kiếm Minimax.

1.1 Cây trò chơi và tìm kiếm trên cây trò chơi.

Trong chương này chúng ta chỉ quan tâm nghiên cứu các trò chơi có hai người tham gia, chẳng hạn các loại cờ (cờ vua, cờ tướng, cờ ca rô...). Một người chơi được gọi là Trắng, đối thủ của anh ta được gọi là Đen. Mục tiêu của chúng ta là nghiên cứu chiến lược chọn nước đi cho Trắng (Máy tính cầm quân Trắng).

Chúng ta sẽ xét các trò chơi hai người với các đặc điểm sau. Hai người chơi thay phiên nhau đưa ra các nước đi tuân theo các luật đi nào đó, các luật này là như nhau cho cả hai người. Điển hình là cờ vua, trong cờ vua hai người chơi có

thể áp dụng các luật đi con tốt, con xe, ... để đưa ra nước đi. Luật đi con tốt Trắng xe Trắng, ... cũng như luật đi con tốt Đen, xe Đen, ... Một đặc điểm nữa là hai người chơi đều được biết thông tin đầy đủ về các tình thế trong trò chơi (không như trong chơi bài, người chơi không thể biết các người chơi khác còn những con bài gì). Vấn đề chơi cờ có thể xem như vấn đề tìm kiếm nước đi, tại mỗi lần đến lượt mình, người chơi phải tìm trong số rất nhiều nước đi hợp lệ (tuân theo đúng luật đi), một nước đi tốt nhất sao cho qua một dãy nước đi đã thực hiện, anh ta giành phần thắng. Tuy nhiên vấn đề tìm kiếm ở đây sẽ phức tạp hơn vấn đề tìm kiếm mà chúng ta đã xét trong các chương trước, bởi vì ở đây có đối thủ, người chơi không biết được đối thủ của mình sẽ đi nước nào trong tương lai. Sau đây chúng ta sẽ phát biểu chính xác hơn vấn đề tìm kiếm này.

Vấn đề chơi cờ có thể xem như vấn đề tìm kiếm trong không gian trạng thái. Mỗi trạng thái là một tình thế (sự bố trí các quân của hai bên trên bàn cờ).

- Trạng thái ban đầu là sự sắp xếp các quân cờ của hai bên lúc bắt đầu cuộc chơi.
- Các toán tử là các nước đi hợp lệ.
- Các trạng thái kết thúc là các tình thế mà cuộc chơi dừng, thường được xác định bởi một số điều kiện dừng nào đó.
- Một hàm kết cuộc (payoff function) ứng mỗi trạng thái kết thúc với một giá trị nào đó. Chẳng hạn như cờ vua, mỗi trạng thái kết thúc chỉ có thể là thắng, hoặc thua (đối với Trắng) hoặc hòa. Do đó, ta có thể xác định hàm kết cuộc là hàm nhận giá trị 1 tại các trạng thái kết thúc là thắng (đối với Trắng), -1 tại các trạng thái kết thúc là thua (đối với Trắng) và 0 tại các trạng thái kết thúc hòa. Trong một số trò chơi khác, chẳng hạn trò chơi tính điểm, hàm kết cuộc có thể nhận giá trị nguyên trong khoảng [-k, k] với k là một số nguyên dương nào đó.

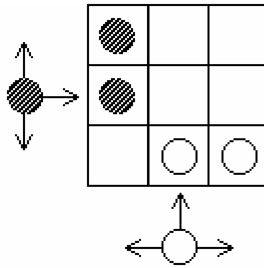
Như vậy vấn đề của Trắng là, tìm một dãy nước đi sao cho xen kẽ với các nước đi của Đen tạo thành một đường đi từ trạng thái ban đầu tới trạng thái kết thúc là thắng cho Trắng.

Để thuận lợi cho việc nghiên cứu các chiến lược chọn nước đi, ta biểu diễn không gian trạng thái trên dưới dạng cây trò chơi.

Cây trò chơi

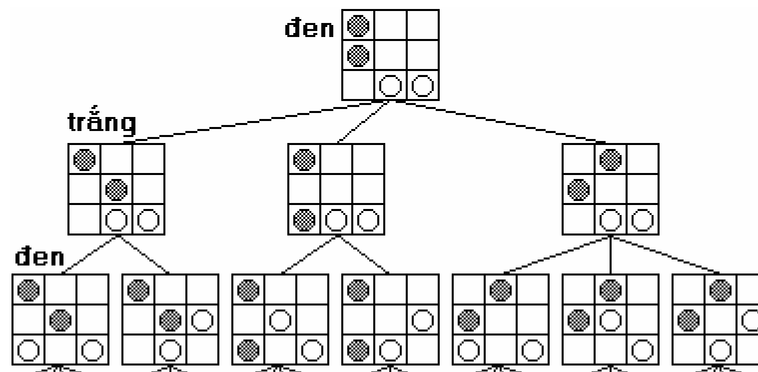
Cây trò chơi được xây dựng như sau. Gốc của cây ứng với trạng thái ban đầu. Ta sẽ gọi đỉnh ứng với trạng thái mà Trắng (Đen) đưa ra nước đi là đỉnh Trắng (Đen). Nếu một đỉnh là Trắng (Đen) ứng với trạng thái u, thì các đỉnh con của nó là tất cả các đỉnh biểu diễn trạng thái v, v nhận được từ u do Trắng (Đen) thực hiện nước đi hợp lệ nào đó. Do đó, trên cùng một mức của cây các đỉnh đều là Trắng hặc đều là Đen, các lá của cây ứng với các trnạg thái kết thúc.

Ví dụ: Xét trò chơi Dodgen (được tạo ra bởi Colin Vout). Có hai quân Trắng và hai quân Đen, ban đầu được xếp vào bàn cờ 3*3 (Hình vẽ). Quân Đen có thể đi tới ô trống ở bên phải, ở trên hoặc ở dưới. Quân Trắng có thể đi tới trống ở bên trái, bên phải, ở trên. Quân Đen nếu ở cột ngoài cùng bên phải có thể đi ra khỏi bàn cờ, quân Trắng nếu ở hàng trên cùng có thể đi ra khỏi bàn cờ. Ai đưa hai quân của mình ra khỏi bàn cờ trước sẽ thắng, hoặc tạo ra tình thế bắt đối phương không đi được cũng sẽ thắng.



Hình 4.1 Trò chơi Dodgem.

Giả sử Đen đi trước, ta có cây trò chơi được biểu diễn như trong hình 4.2.



Hình 4.2 Cây trò chơi Dodgem với Đen đi trước.

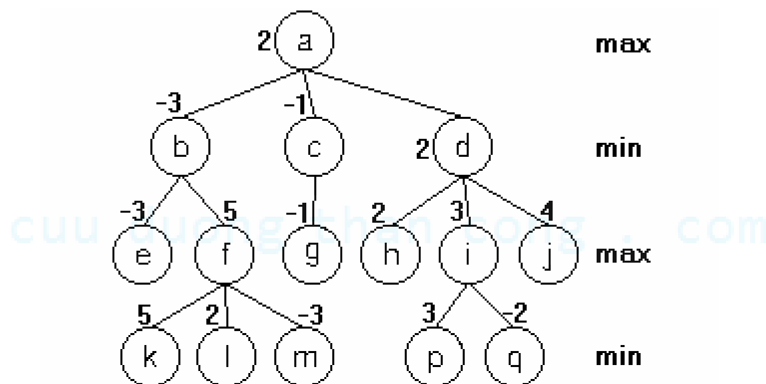
1.2 Chiến lược Minimax

Quá trình chơi cờ là quá trình Trắng và Đen thay phiên nhau đưa ra quyết định, thực hiện một trong số các nước đi hợp lệ. Trên cây trò chơi, quá trình đó sẽ tạo ra đường đi từ gốc tới lá. Giả sử tới một thời điểm nào đó, đường đi đã dẫn tới đỉnh u. Nếu u là đỉnh Trắng (Đen) thì Trắng (Đen) cần chọn đi tới một trong các đỉnh Đen (Trắng) v là con của u. Tại đỉnh Đen (Trắng) v mà Trắng (Đen) vừa chọn, Đen (Trắng) sẽ phải chọn đi tới một trong các đỉnh Trắng (Đen) w là con của v. Quá trình trên sẽ dừng lại khi đạt tới một đỉnh là lá của cây.

Giả sử Trắng cần tìm nước đi tại đỉnh u. Nước đi tối ưu cho Trắng là nước đi dẫn tới đỉnh con của v là đỉnh tốt nhất (cho Trắng) trong số các đỉnh con của u. Ta cần giả thiết rằng, đến lượt đối thủ chọn nước đi từ v, Đen cũng sẽ chọn nước đi tốt nhất cho anh ta. Như vậy, để chọn nước đi tối ưu cho Trắng tại đỉnh u, ta cần phải xác định giá trị các đỉnh của cây trò chơi gốc u. Giá trị của các đỉnh lá (ứng

với các trạng thái kết thúc) là giá trị của hàm kết cuộc. Đỉnh có giá trị càng lớn càng tốt cho Trắng, đỉnh có giá trị càng nhỏ càng tốt cho Đen. Để xác định giá trị các đỉnh của cây trò chơi gốc u, ta đi từ mức thấp nhất lên gốc u. Giả sử v là đỉnh trong của cây và giá trị các đỉnh con của nó đã được xác định. Khi đó nếu v là đỉnh Trắng thì giá trị của nó được xác định là giá trị lớn nhất trong các giá trị của các đỉnh con. Còn nếu v là đỉnh Đen thì giá trị của nó là giá trị nhỏ nhất trong các giá trị của các đỉnh con.

Ví dụ: Xét cây trò chơi trong hình 4.3, gốc a là đỉnh Trắng. Giá trị của các đỉnh là số ghi cạnh mỗi đỉnh. Đỉnh i là Trắng, nên giá trị của nó là max(3,-2) = 3,



Hình 4.3 Gán giá trị cho các đỉnh của cây trò chơi.

đỉnh d là đỉnh Đen, nên giá trị của nó là min(2, 3, 4) = 2.

Việc gán giá trị cho các đỉnh được thực hiện bởi các hàm đệ qui MaxVal và MinVal. Hàm MaxVal xác định giá trị cho các đỉnh Trắng, hàm MinVal xác định giá trị cho các đỉnh Đen.

function *MaxVal(u)***;**

begin

if *u là đỉnh kết thúc* **then** *MaxVal(u)* ← *f(u)*

else *MaxVal(u)* ← *max***{***MinVal(v)* **|** *v là đỉnh con của u***}**

end;

function *MinVal(u)***;**

begin

if *u là đỉnh kết thúc* **then** *MinVal(u) ← f(u)*

else *MinVal(u) ← min***{***MaxVal(v)* **|** *v là đỉnh con của u***}**

end;

Trong các hàm đệ quy trên, f(u) là giá trị của hàm kết cuộc tại đỉnh kết thúc u. Sau đây là thủ tục chọn nước đi cho trắng tại đỉnh u. Trong thủ tục Minimax(u,v), v là biến lưu lại trạng thái mà Trắng đã chọn đi tới từ u.

procedure *Minimax(u, v)***;**

begin

*val ← -∞***;**

for *mỗi w là đỉnh con của u* **do**

if *val <= MinVal(w)* **then**

{*val ← MinVal(w)***;** *v ← w***}**

end;

Thủ tục chọn nước đi như trên gọi là chiến lược Minimax, bởi vì Trắng đã chọ được nước đi dẫn tới đỉnh con có giá trị là max của các giá trị các đỉnh con, và Đen đáp lại bằng nước đi tới đỉnh có giá trị là min của các giá trị các đỉnh con.

Thuật toán Minimax là thuật toán tìm kiếm theo độ sâu, ở đây ta đã cài đặt thuật toán Minimax bởi các hàm đệ quy. Bạn đọc hãy viết thủ tục không đệ quy thực hiện thuật toán này.

Về mặt lí thuyết, chiến lược Minimax cho phép ta tìm được nước đi tối ưu cho Trắng. Song nó không thực tế, chúng ta sẽ không có đủ thời gian để tính được nước đi tối ưu. Bởi vì thuật toán Minimax đòi hỏi ta phải xem xét toàn bộ các đỉnh của cây trò chơi. Trong các trò chơi hay, cây trò chơi là cực kỳ lớn. Chẳng hạn, đối với cờ vua, chỉ tính đến độ sâu 40, thì cây trò chơi đã có khoảng 10^{120} đỉnh! Nếu cây có độ cao m, và tại mỗi đỉnh có b nước đi thì độ phức tạp về thời gian của thuật toán Minimax là $O(b^m)$.

Để có thể tìm ra nhanh nước đi tốt (không phải là tối ưu) thay cho việc sử dụng hàm kết cuộc và xem xét tất cả các khả năng dẫn tới các trạng thái kết thúc, chúng ta sẽ sử dụng hàm đánh giá và chỉ xem xét một bộ phận của cây trò chơi.

Hàm đánh giá

Hàm đánh giá eval ứng với mỗi trạng thái u của trò chơi với một giá trị số eval(u), giá trị này là sự đánh giá "độ lợi thế" của trạng thái u. Trạng thái u càng thuận lợi cho Trắng thì eval(u) là số dương càng lớn; u càng thuận lợi cho Đen thì eval(u) là số âm càng nhỏ; eval(u) ≈ 0 đối với trạng thái không lợi thế cho ai cả.

Chất lượng của chương trình chơi cờ phụ thuộc rất nhiều vào hàm đánh giá. Nếu hàm đánh giá cho ta sự đánh giá không chính xác về các trạng thái, nó có thể hướng dẫn ta đi tới trạng thái được xem là tốt, nhưng thực tế lại rất bất lợi cho ta. Thiết kế một hàm đánh giá tốt là một việc khó, đòi hỏi ta phải quan tâm đến nhiều

nhân tố: các quân còn lại của hai bên, sự bố trí của các quân đó, ... ở đây có sự mâu thuẫn giữa độ chính xác của hàm đánh giá và thời gian tính của nó. Hàm đánh giá chính xác sẽ đòi hỏi rất nhiều thời gian tính toán, mà người chơi lại bị giới hạn bởi thời gian phải đưa ra nước đi.

Ví dụ 1: Sau đây ta đưa ra một cách xây dựng hàm đánh giá đơn giản cho cờ vua. Mỗi loại quân được gán một giá trị số phù hợp với "sức mạnh" của nó. Chẳng hạn, mỗi tốt Trắng (Đen) được cho 1 (-1), mã hoặc tượng Trắng (Đen) được cho 3 (-3), xe Trắng (Đen) được cho 5 (-5) và hoàng hậu Trắng (Đen) được cho 9 (-9). Lấy tổng giá trị của tất cả các quân trong một trạng thái, ta sẽ được giá trị đánh giá của trạng thái đó. Hàm đánh giá như thế được gọi là hàm tuyến tính có trọng số, vì nó có thể biểu diễn dưới dạng:

$$s_1w_1 + s_2w_2 + . . . + s_nw_n.$$

Trong đó, w_i là giá trị mỗi loại quân, còn s_i là số quân loại đó. Trong cách đánh giá này, ta đã không tính đến sự bố trí của các quân, các mối tương quan giữa chúng.

Ví dụ 2: Bây giờ ta đưa ra một cách đánh giá các trạng thái trong trò chơi Dodgem. Mỗi quân Trắng ở một vị trí trên bàn cờ được cho một giá trị tương ứng trong bảng bên trái hình 4.4. Còn mỗi quân Đen ở một vị trí sẽ được cho một giá trị tương ứng trong bảng bên phải hình 4.4:

30	35	40
15	20	25
0	5	10

Giá trị quân Trắng.

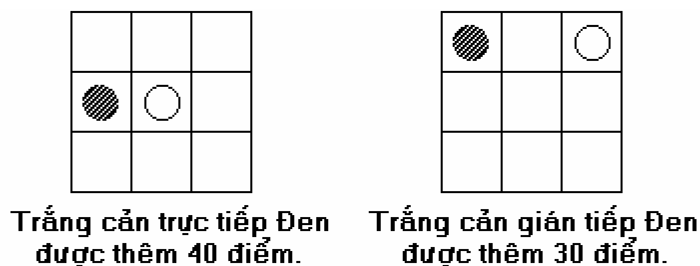
-10	-25	-40
-5	-20	-35
0	-15	-30

Giá trị quân Đen.

Hình 4.4 Đánh giá các quân trong trò chơi Dodgem.

Ngoài ra, nếu quân Trắng cản trực tiếp một quân Đen, nó được thêm 40 điểm, nếu cản gián tiếp nó được thêm 30 điểm (Xem hình 4.5). Tương tự, nếu quân Đen cản trực tiếp quân Trắng nó được thêm -40 điểm, còn cản gián tiếp nó được thêm -30 điểm.

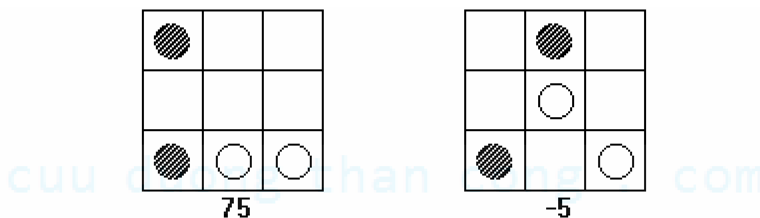
áp dụng các qui tắc trên, ta tính được giá trị của trạng thái ở bên trái hình 4.6 là 75, giá trị của trạng thái bên phải hình vẽ là -5.



Hình 4.5 Đánh giá sự tương quan giữa quân Trắng và Đen.

Trong cánh đánh giá trên, ta đã xét đến vị trí của các quân và mối tương quan giữa các quân.

Một cách đơn giản để hạn chế không gian tìm kiếm là, khi cần xác định nước đi cho Trắng tại u, ta chỉ xem xét cây trò chơi gốc u tới độ cao h nào đó. áp dụng thủ tục Minimax cho cây trò chơi gốc u, độ cao h và sử dụng giá trị của hàm đánh



Hình 4.6 Giá trị của một số trạng thái trong trò chơi Dodgem.

giá cho các lá của cây đó, chúng ta sẽ tìm được nước đi tốt cho Trắng tại u.

1.3 Phương pháp cắt cụt alpha - beta

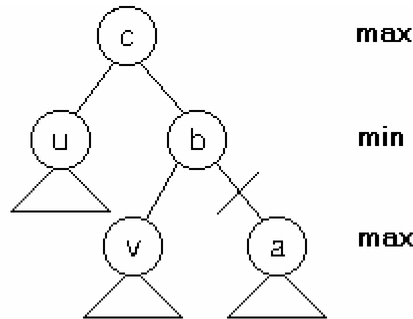
Trong chiến lược tìm kiếm Minimax, để tìm kiếm nước đi tốt cho Trắng tại trạng thái u, cho dù ta hạn chế không gian tìm kiếm trong phạm vi cây trò chơi gốc u với độ cao h, thì số đỉnh của cây trò chơi này cũng còn rất lớn với $h \geq 3$. Chẳng hạn, trong cờ vua, nhân tố nhánh trong cây trò chơi trung bình khoảng 35, thời gian đòi hỏi phải đưa ra nước đi là 150 giây, với thời gian này trên máy tính thông thường chương trình của bạn chỉ có thể xem xét các đỉnh trong độ sâu 3 hoặc 4. Một người chơi cờ trình độ trung bình cũng có thể tính trước được 5, 6 nước hoặc hơn nữa, và do đó chương trình của bạn mới đạt trình độ người mới tập chơi!

Khi đánh giá đỉnh u tới độ sâu h, một thuật toán Minimax đòi hỏi ta phải đánh giá tất cả các đỉnh của cây gốc u tới độ sâu h. Song ta có thể giảm bớt số đỉnh cần phải dánh giá mà vẫn không ảnh hưởng gì đến sự đánh giá đỉnh u. Phương pháp cắt cụt alpha-beta cho phép ta cắt bỏ các nhánh không cần thiết cho sự đánh giá đỉnh u.

Tư tưởng của kỹ thuật cắt cụt alpha-beta là như sau: Nhớ lại rằng, chiến lược tìm kiếm Minimax là chiến lược tìm kiếm theo độ sâu. Giả sử trong quá trính tìm kiếm ta đi xuống đỉnh a là đỉnh Trắng, đỉnh a có người anh em v đã được đánh giá. Giả sử cha của đỉnh a là b và b có người anh em u dã được đánh giá, và giả sử cha của b là c (Xem hình 4.7). Khi đó ta có giá trị đỉnh c (đỉnh Trắng) ít nhất là giá trị của u, giá trị của đỉnh b (đỉnh Đen) nhiều nhất là giá trị v. Do đó, nếu eval(u) > eval(v), ta không cần đi xuống để đánh giá đỉnh a nữa mà vẫn không ảnh hưởng gì dến đánh giá đỉnh c. Hay nói cách khác ta có thể cắt bỏ cây con gốc a. Lập luận tương tự cho trường hợp a là đỉnh Đen, trong trường hợp này nếu eval(u) < eval(v) ta cũng có thể cắt bỏ cây con gốc a.

Để cài đặt kỹ thuật cắt cụt alpha-beta, đối với các đỉnh nằm trên đường đi từ gốc tới đỉnh hiện thời, ta sử dụng tham số α để ghi lại giá trị lớn nhất trong các giá

trị của các đỉnh con đã đánh giá của một đỉnh Trắng, còn tham số β ghi lại giá trị nhỏ nhất trong các đỉnh con đã đánh giá của một đỉnh Đen. Giá trị của α và β sẽ được cập nhật trong quá trình tìm kiếm. α và β được sử dụng như các biến địa phương trong các hàm MaxVal(u, α, β) (hàm xác định giá trị của đỉnh Trắng u) và



Hình 4.7 Cắt bỏ cây con gốc a, nếu eval(u) > eval(v).

Minval(u, α, β) (hàm xác định giá trị của đỉnh Đen u).

function *MaxVal(u, α, β)*;

begin

if *u là lá của cây hạn chế hoặc u là đỉnh kết thúc*

then *MaxVal ← eval(u)*

else for *mỗi đỉnh v là con của u* **do**

{*α ← max[α, MinVal(v, α, β)]*;

// Cắt bỏ các cây con từ các đỉnh v còn lại

if *α ≥ β* **then exit}**;

MaxVal ← α;

end;

function *MinVal(u, α, β)***;**

begin

if *u là lá của cây hạn chế hoặc u là đỉnh kết thúc*

then *MinVal* $\leftarrow$ *eval(u)*

else for *mỗi đỉnh v là con của u* **do**

{$\beta \leftarrow min[\beta, MaxVal(v, \alpha, \beta)]$**;**

// Cắt bỏ các cây con từ các đỉnh v còn lại

if $\alpha \geq \beta$ **then exit};**

MinVal $\leftarrow \beta$**;**

end;

Thuật toán tìm nước đi cho Trắng sử dụng kỹ thuật cắt cụt alpha-beta, được cài đặt bởi thủ tục Alpha_beta(u,v), trong đó v là tham biến ghi lại đỉnh mà Trắng cần đi tới từ u.

procedure *Alpha_beta(u,v)***;**

begin

$\alpha \leftarrow -\infty$**;**

$\beta \leftarrow \infty$**;**

for *mỗi đỉnh w là con của u* **do**

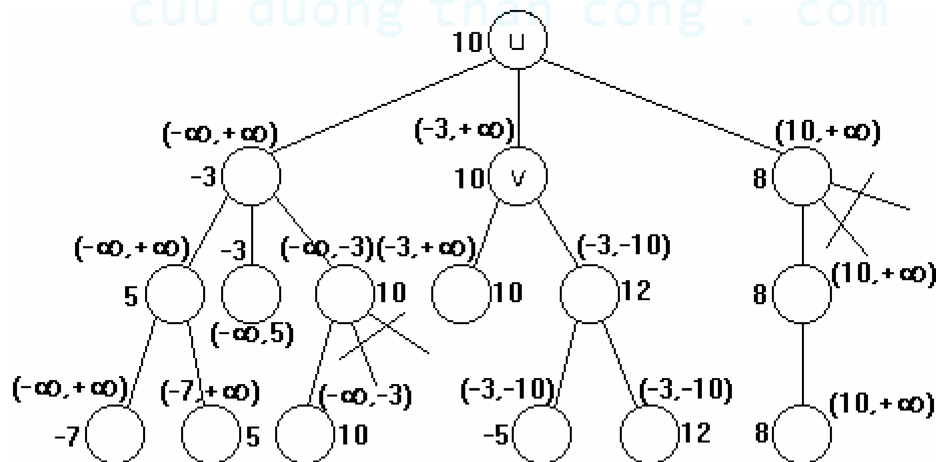
if $\alpha \le MinVal(w, \alpha, \beta)$ **then**

{$\alpha \leftarrow MinVal(w, \alpha, \beta)$;

$v \leftarrow w$;}

end;

Ví dụ. Xét cây trò chơi gốc u (đỉnh Trắng) giới hạn bởi độ cao h = 3 (hình 4.8). Số ghi cạnh các lá là giá trị của hàm đánh giá. áp dụng chiến lược Minimax và kỹ thuật cắt cụt, ta xác định được nước đi tốt nhất cho Trắng tại u, đó là nước đi dẫn tới đỉnh v có giá trị 10. Cạnh mỗi đỉnh ta cũng cho giá trị của cặp tham số (α, β). Khi gọi các hàm MaxVal và MinVal để xác định giá trị của đỉnh đó. Các nhánh bị cắt bỏ được chỉ ra trong hình:



Hình 4.8 Xác định giá trị các đỉnh bằng kỹ thuật cắt cụt.

Phần II:

Tri thức và lập luận

Chương V:

Logic mệnh đề

Trong chương này chúng ta sẽ trình bày các đặc trưng của ngôn ngữ biểu diễn tri thức. Chúng ta sẽ nghiên cứu logic mệnh đề, một ngôn ngữ biểu diễn tri thức rất đơn giản, có khả năng biểu diễn hẹp, nhưng thuận lợi cho ta làm quen với nhiều khái niệm quan trọng trong logic, đặc biệt trong logic vị từ cấp một sẽ được nghiên cứu trong các chương sau.

5.1. Biểu diễn tri thức

Con người sống trong môi trường có thể nhận thức được thế giới nhờ các giác quan (tai, mắt và các bộ phận khác), sử dụng các tri thức tích luỹ được và nhờ khả năng lập luận, suy diễn, con người có thể đưa ra các hành động hợp lý cho công việc mà con người đang làm. Một mục tiêu của Trí tuệ nhân tạo ứng dụng là thiết kế các *tác nhân thông minh* (intelligent agent) cũng có khả năng đó như con người. Chúng ta có thể hiểu tác nhân thông minh là bất cứ cái gì có thể nhận thức được môi trường thông qua các bộ cảm nhận (sensors) và đưa ra hành động hợp lý đáp ứng lại môi trường thông qua bộ phận hành động (effectors). Các robots, các softbot (software robot), các hệ chuyên gia,... là các ví dụ về tác nhân thông minh. Các tác nhân thông minh cần phải có tri thức về thế giới hiện thực mới có thể đưa ra các quyết định đúng đắn.

Thành phần trung tâm của các *tác nhân dựa trên tri thức* (knowledge-based agent), còn được gọi là *hệ dựa trên tri thức* (knowledge-based system) hoặc đơn giản là hệ tri thức, là cơ sở tri thức. Cơ sở tri thức (CSTT) là một tập hợp các tri thức được biểu diễn dưới dạng nào đó. Mỗi khi nhận được các thông tin đưa vào, tác nhân cần có khả năng suy diễn để đưa ra các câu trả lời, các hành động hợp lý, đúng đắn. Nhiệm vụ này được thực hiện bởi bộ suy diễn. Bộ suy diễn là thành phần cơ bản khác của các hệ tri thức. Như vậy hệ tri thức bảo trì một CSTT và được trang bị một thủ tục suy diễn. Mỗi khi tiếp nhận được các sự kiện từ môi trường, thủ tục suy diễn thực hiện quá trình liên kết các sự kiện với các tri thức trong CSTT để rút ra các câu trả lời, hoặc các hành động hợp lý mà tác nhân cần thực hiện. Đương nhiên là, khi ta thiết kế một tác nhân giải quyết một vấn đề nào đó thì CSTT sẽ chứa các tri thức về miền đối tượng cụ thể đó. Để máy tính có thể

sử dụng được tri thức, có thể xử lý tri thức, chúng ta cần biểu diễn tri thức dưới dạng thuận tiện cho máy tính. Đó là mục tiêu của biểu diễn tri thức.

Tri thức được mô tả dưới dạng các câu trong *ngôn ngữ biểu diễn tri thức*. Mỗi câu có thể xem như sự mã hóa của một sự hiểu biết của chúng ta về thế giới hiện thực. Ngôn ngữ biểu diễn tri thức (cũng như mọi ngôn ngữ hình thức khác) gồm hai thành phần cơ bản là *cú pháp* và *ngữ nghĩa*.

- Cú pháp của một ngôn ngữ bao gồm các ký hiệu về các quy tắc liên kết các ký hiệu (các luật cú pháp) để tạo thành các câu (công thức) trong ngôn ngữ. Các câu ở đây là biểu diễn ngoài, cần phân biệt với biểu diễn bên trong máy tính. Các câu sẽ được chuyển thành các cấu trúc dữ liệu thích hợp được cài đặt trong một vùng nhớ nào đó của máy tính, đó là biểu diễn bên trong. Bản thân các câu chưa chứa đựng một nội dung nào cả, chưa mang một ý nghĩa nào cả.

- Ngữ nghĩa của ngôn ngữ cho phép ta xác định ý nghĩa của các câu trong một miền nào đó của thế giới hiện thực. Chẳng hạn, trong ngôn ngữ các biểu thức số học, dãy ký hiệu (x+y)*z là một câu viết đúng cú pháp. Ngữ nghĩa của ngôn ngữ này cho phép ta hiểu rằng, nếu x, y, z, ứng với các số nguyên, ký hiệu + ứng với phép toán cộng, còn * ứng với phép chia, thì biểu thức (x+y)*z biểu diễn quá trình tính toán: lấy số nguyên x cộng với số nguyên y, kết quả được nhân với số nguyên z.

- Ngoài hai thành phần cú pháp và ngữ nghĩa, ngôn ngữ biểu diễn tri thức cần được cung cấp *cơ chế suy diễn*. Một luật suy diễn (rule of inference) cho phép ta suy ra một công thức từ một tập nào đó các công thức. Chẳng hạn, trong logic mệnh đề, luật modus ponens từ hai công thức A và A⇒B suy ra công thức B. Chúng ta sẽ hiểu *lập luận* hoặc *suy diễn* là một quá trình áp dụng các luật suy diễn để từ các tri thức trong cơ sở tri thức và các sự kiện ta nhận được các tri thức mới. Như vậy chúng ta xác định:

1.1 Ngôn ngữ biểu diễn tri thức = Cú pháp + Ngữ nghĩa + Cơ chế suy diễn.

1.2 Một ngôn ngữ biểu diễn tri thức tốt cần phải có khả năng biểu diễn rộng, tức là có thể mô tả được mọi điều mà chúng ta muốn nói. Nó cần phải hiệu quả theo nghĩa là, để đi tới các kết luận, thủ tục suy diễn đòi hỏi ít thời gian tính toán và ít không gian nhớ. Người ta cũng mong muốn ngôn ngữ biểu diễn tri thức gần với ngôn ngữ tự nhiên.

1.3 Trong sách này, chúng ta sẽ tập trung nghiên cứu logic vị từ cấp một (first-order predicate logic hoặc first-order predicate calculus) - một ngôn ngữ biểu diễn tri thức, bởi vì logic vị từ cấp một có khả năng biểu diễn tương đối tốt, và hơn nữa nó là cơ sở cho nhiều ngôn ngữ biểu diễn tri thức khác, chẳng hạn toán hoàn cảnh (situation calculus) hoặc logic thời gian khoảng cấp một (first-order interval tempral logic). Nhưng trước hết chúng ta sẽ nghiên cứu logic mệnh đề (propositional logic hoặc propositional calculus). Nó là ngôn ngữ rất đơn giản, có khả năng biểu diễn hạn chế, song thuận tiện cho ta đưa vào nhiều khái niệm quan trọng trong logic.

5.2. Cú pháp và ngữ nghĩa của logic mệnh đề.

5.2.1 Cú pháp:

Cú pháp của logic mệnh đề rất đơn giản, nó cho phép xây dựng nên các công thức. Cú pháp của logic mệnh đề bao gồm tập các ký hiệu và tập các luật xây dựng công thức.

1. Các ký hiệu

- Hai hằng logic True và False.
- Các ký hiệu mệnh đề (còn được gọi là các biến mệnh đề): P, Q,...
- Các kết nối logic $\wedge$, $\vee$, $\rceil$, $\Rightarrow$, $\Leftrightarrow$.
- Các dấu mở ngoặc (và đóng ngoặc).

2. Các quy tắc xây dựng các công thức

- Các biến mệnh đề là công thức.
- Nếu A và B là công thức thì:

$(A \wedge B)$ (đọc “A hội B” hoặc “A và B”)

$(A \vee B)$ (đọc “A tuyển B” hoặc “A hoặc B”)

$(\rceil A)$ (đọc “phủ định A”)

$(A \Rightarrow B)$ (đọc “A kéo theo B” hoặc “nếu A thì B”)

$(A \Leftrightarrow B)$ (đọc “A và B kéo theo nhau”)

là các công thức.

Sau này để cho ngắn gọn, ta sẽ bỏ đi các cặp dấu ngoặc không cần thiết. Chẳng hạn, thay cho $((A \lor B) \land C)$ ta sẽ viết là $(A \lor B) \land C$.

Các công thức là các ký hiệu mệnh đề sẽ được gọi là các *câu đơn* hoặc *câu phân tử*. Các công thức không phải là câu đơn sẽ được gọi là câu phức hợp. Nếu P là ký hiệu mệnh đề thì P và TP được gọi là *literal*, P là *literal dương*, còn TP là *literal âm*. Câu phức hợp có dạng $A_1 \lor ... \lor A_m$ trong đó A_i là các literal sẽ được gọi là *câu tuyển* (clause).

5.2.2 **Ngữ nghĩa:**

Ngữ nghĩa của logic mệnh đề cho phép ta *xác định* thiết lập ý nghĩa của các công thức trong thế giới hiện thực nào đó. Điều đó được thực hiện bằng cách kết hợp mệnh đề với sự kiện nào đó trong thế giới hiện thực. Chẳng hạn, ký hiệu mệnh đề P có thể ứng với sự kiện "Paris là thủ đô nước Pháp" hoặc bất kỳ một sự kiện nào khác. Bất kỳ một sự kết hợp các kí hiệu mệnh đề với các sự kiện trong thế giới thực được gọi là một *minh họa* (interpretation). Chẳng hạn minh họa của kí hiệu mệnh đề P có thể là một sự kiện (mệnh đề) "Paris là thủ đô nước Pháp ". Một sự kiện chỉ có thể đúng hoặc sai. Chẳng hạn, sự kiện "Paris là thủ đô nước Pháp " là đúng, còn sự kiện "Số Pi là số hữu tỉ " là sai.

Một cách chính xác hơn, cho ta hiểu một minh họa là một cách gán cho mỗi ký hiệu mệnh đề một giá trị chân lý **True** hoặc **False.** Trong một minh họa, nếu kí hiệu mệnh đề P được gán giá trị chân lý **True/False** (P <-**True**/ P<-**False**) thì ta nói mệnh đề P **đúng/sai** trong minh họa đó. Trong một minh họa, ý nghĩa của các câu phức hợp được xác định bởi ý nghĩa của các kết nối logic. Chúng ta xác định ý nghĩa của các kết nối logic trong các bảng chân lý (xem hình 5.1)

P	Q	lP	P∧ Q	P v Q	P=> Q	P<=>Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

Hình 5.1 Bảng chân lý của các kết nối logic

ý nghĩa của các kết nối logic ∧, v và l được xác định như các từ "**và**","**hoặc là**" và "**phủ định**" trong ngôn ngữ tự nhiên. Chúng ta cần phải giải thích thêm về ý nghĩa của phép kéo theo P => Q (P kéo theo Q), P là giả thiết, còn Q là kết luận. Trực quan cho phép ta xem rằng, khi P là đúng và Q là đúng thì câu "P kéo theo Q " là đúng, còn khi P là đúng Q là sai thì câu "P kéo theo Q" là sai.

Nhưng nếu P sai và Q đúng , hoặc P sai Q sai thì “P kéo theo Q” là đúng hay sai ? Nếu chúng ta xuất phát từ giả thiết sai, thì chúng ta không thể khẳng định gì về kết luận. Không có lý do gì để nói rằng, nếu P sai và Q đúng hoặc P sai và Q sai thì “P kéo theo Q” là sai. Do đó trong trường hợp P sai thì “P kéo theo Q ” là đúng dù Q là đúng hay Q là sai.

Bảng chân lý cho phép ta xác định ngẫu nhiên các câu phức hợp. Chẳng hạn ngữ nghĩa của các câu P∧Q trong minh họa {P <- **True** , Q<- **False** } là **False**. Việc xác định ngữ nghĩa của một câu (P v Q) ∧ lS trong một minh họa được tiến hành như sau: đầu tiên ta xác định giá trị chân lý của P v Q và l S , sau đó ta sử dụng bảng chân lý ∧ để xác định giá trị (PvQ) ∧lS

- Một công thức được gọi là *thoả được* (satisfiable) nếu nó đúng trong một minh họa nào đó. Chẳng hạn công thức (PvQ) ∧lS là thoả được, vì nó có giá trị **True** trong minh họa {P <- **True**, Q<-**False**, S<- **True**}.

- Một công thức được gọi là *vững chắc*_(valid hoặc tautology) nếu nó đúng trong mọi minh họa chẳng hạn câu P vlP là vững chắc

- Một công thức được gọi là *không thoả được* , nếu nó là sai trong mọi minh họa. Chẳng hạn công thức P ∧ lP.

Chúng ta sẽ gọi một *mô hình* (modul) của một công thức là một minh họa sao cho công thức là đúng trong minh họa này. Như vậy một công thức *thoả được* là công thức có một mô hình. Chẳng hạn, minh họa {P <- **False** , Q <- **False** , S<- **True** } là một mô hình của công thức (P =>Q) ∧ S .

Bằng cách lập bảng chân lý (*phương pháp bảng chân lý*) là ta có thể xác định được một công thức có *thoả được* hay không. Trong bảng này, mỗi biến mệnh đề đứng đầu với một cột, công thức cần kiểm tra đứng đầu một cột, mỗi dòng tương ứng với một minh họa. Chẳng hạn hình 5.2 là bảng chân lý cho công thức (P=>Q) ∧S. Trong bảng chân lý này ta cần đưa vào các cột phụ ứng với các công thức con của các công thức cần kiểm tra để việc tính giá trị của công thức này được dễ dàng. Từ bảng chân lý ta thấy rằng công thức (P=>Q) ∧S là *thoả được* nhưng không *vững chắc* .

P	Q	S	P=>Q	(P=>Q) ∧S
False	False	False	True	False
False	False	True	True	True
False	True	False	True	False
False	True	True	True	True
True	False	False	False	False

True	False	True	False	False
True	True	False	True	False
True	True	True	True	True

Hình 5.2 Bảng chân lý cho công thức (P=>Q) ∧S

Cần lưu ý rằng, một công thức chứa n biến, thì số các minh họa của nó là 2^n , tức là bảng chân lý có 2^n dòng. Như vậy việc kiểm tra một công thức có *thoả được* hay không bằng phương pháp bảng chân lý, đòi hỏi thời gian mũ. Cook (1971) đã chứng minh rằng, vấn đề kiểm tra một công thức trong logic mệnh đề có *thoả được* hay không là vấn đề NP-đầy đủ.

Chúng ta sẽ nói rằng (*thoả được, không thoả được*) nếu hội của chúng $G_1 \wedge ....\wedge G_m$ là *vững chắc* (*thoả được, không thoả được*). Một mô hình của tập công thức G là mô hình của tập công thức $G_1 \wedge ....\wedge G_m$.

5.3 Dạng chuẩn tắc

Trong mục này chúng ta sẽ xét việc chuẩn hóa các công thức, đưa các công thức về dạng thuận lợi cho việc lập luận, suy diễn. Trước hết ta sẽ xét các phép biến đổi tương đương. Sử dụng các phép biến đổi này, ta có thể đưa một công thức bất kỳ về các dạng chuẩn tắc.

5.3.1 Sự tương đương của các công thức

Hai công thức A và B được xem là *tương đương* nếu chúng có cùng một giá trị chân lý trong mọi minh họa. Để chỉ A tương đương với B ta viết A≡ B bằng phương pháp bảng chân lý, dễ dàng chứng minh được sự tương đương của các công thức sau đây :

- A=>B ≡ lA v B
- A< = > B ≡ (A=>B) ∧ (B=>A)
- l(lA) ≡ A

1.4 Luật De Morgan

- l(A v B) ≡ lA ∧ lB

- l(A ∧ B) ≡ lA v lB

1.5 Luật giao hoán

- A v B ≡ B v A

- $A \wedge B \equiv B \wedge A$

1.6 Luật kết hợp

- $(A \vee B) \vee C \equiv A \vee (B \vee C)$
- $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$

1.7 Luật phân phối

- $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

5.3.2 *Dạng chuẩn tắc :*

Các công thức tương đương có thể xem như các biểu diễn khác nhau của cùng một sự kiện. Để dễ dàng viết các chương trình máy tính thao tác trên các công thức, chúng ta sẽ chuẩn hóa các công thức, đưa chúng về dạng biểu diễn chuẩn được gọi là *dạng chuẩn hội.* Một công thức ở dạng chuẩn hội, có dạng A_1 v ... .v A_m trong đó các A_i là *literal* . Chúng ta có thể biến đổi một công thức bất kỳ về công thức ở dạng chuẩn hội bằng cách áp dụng các thủ tục sau.

- Bỏ các dấu kéo theo (=>) bằng cách thay (A=>B) bởi (lAvB).
- Chuyển các dấu phủ định (l) vào sát các kết hiệu mệnh đề bằng cách áp dụng luật De Morgan và thay l(lA) bởi A .

- áp dụng luật phân phối, thay các công thức có dạng Av(B∧C) bởi (A v B) ∧ (A v B) .

Ví dụ : Ta chuẩn hóa công thức (P => Q) v l(R v lS) :

(P => Q) v l(R v lS) ≡ (lP v Q) v (lR ∧ S) ≡ ((lP v Q)vlR) ∧ ((lP v Q) v S) ≡ (l P v Q v lR) ∧ (lP v Q v S). Như vậy công thức (P=> Q) v l(R v lS) được đưa về dạng chuẩn hội (lP v Q v lR) ∧ (lP v Q v S).

Khi biểu diễn tri thức bởi các công thức trong logic mệnh đề, cơ sở tri thức là một tập nào đó các công thức. Bằng cách chuẩn hoá các công thức, cơ sở tri thức là một tập nào đó các câu tuyển.

Các câu Horn:

ở trên ta đã chỉ ra, mọi công thức đều có thể đưa về dạng chuẩn hội, tức là các hội của các tuyển, mỗi câu tuyển có dạng

lP_1 v........v lP_m v Q_1 v.....v Q_m

trong đó P_i , Q_i là các ký hiệu mệnh đề (literal dương) câu này tương đương với câu

lP_1 v........v lP_m => v Q_1 v.....v Q_m **????** p1^ .... ^ pm => **Q**

Dạng câu này được gọi là *câu Kowalski* (do nhà logic Kowalski đưa ra năm 1971).

Khi n <=1, tức là câu Kowalski chỉ chứa nhiều nhất một literal dương ta có dạng một câu đặc biệt quan trọng được gọi là *câu Horn* (mang tên nhà logic Alfred Horn năm 1951).

Nếu m>0, n=1, câu Horn có dạng :

$P_1 \wedge ..... \wedge P_m \Rightarrow Q$

Trong đó P_i , Q là các literal dương. Các P_i được gọi là các điều kiện (hoặc giả thiết), còn Q được gọi là kết luận (hoặc hệ quả). Các câu Horn dạng này còn được gọi là các luật ***if ... then*** và được biểu diễn như sau :

If P_1 and ....and P_m *then* Q .

Khi m=0, n=1 câu Horn trở thành câu đơn Q, hay sự kiện Q. Nếu m>0, n=0 câu Horn trở thành dạng lP_1 v......v lP_m hay tương đương l(P_1^...^ P_m). Cần chú ý rằng, không phải mọi công thức đều có thể biểu diễn dưới dạng hội của các câu Horn. Tuy nhiên trong các ứng dụng, cơ sở tri thức thường là một tập nào đó các câu Horn (tức là một tập nào đó các luật if-then).

5.4 Luật suy diễn

Một công thức H được xem là *hệ qủa logic* (logical consequence) của một tập công thức *G* ={G_1,.....,G_m} nếu trong bất kỳ minh họa nào mà {G_1,.....,G_m} đúng thì H cũng đúng, hay nói cách khác bất kỳ một mô hình nào của *G* cũng là mô hình của H.

Khi có một cơ sở tri thức, ta muốn sử dụng các tri thức trong cơ sở này để suy ra tri thức mới mà nó là hệ quả logic của các công thức trong cơ sở tri thức. Điều đó được thực hiện bằng các thực hiện *các luật suy diễn* (rule of inference). Luật suy diễn giống như một thủ tục mà chúng ta sử dụng để sinh ra một công thức mới từ các công thức đã có. Một luật suy diễn gồm hai phần : một tập các điều kiện và một kết luận. Chúng ta sẽ biểu diễn các luật suy diễn dưới dạng "phân số ", trong đó tử số là danh sách các điều kiện, còn mẫu số là kết luận của luật, tức là mẫu số là công thức mới được suy ra từ các công thức ở tử số.

Sau đây là một số luật suy diễn quan trọng trong logic mệnh đề. Trong các luật này α, α_i , β, γ là các công thức :

1. Luật Modus Ponens

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$

Từ một kéo theo và giả thiết của kéo theo, ta suy ra kết luận của nó.

2. Luật Modus Tollens

$$\frac{\alpha \Rightarrow \beta, \rceil\beta}{\rceil\alpha}$$

Từ một kéo theo và phủ định kết luận của nó, ta suy ra phủ định giả thiết của kéo theo.

3. Luật bắc cầu

$$\frac{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\alpha \Rightarrow \gamma}$$

Từ hai kéo theo, mà kết luận của nó là của kéo theo thứ nhất trùng với giả thiết của kéo theo thứ hai, ta suy ra kéo theo mới mà giả thiết của nó là giả thiết của kéo theo thứ nhất, còn kết luận của nó là kết luận của kéo theo thứ hai.

4. Luật loại bỏ hội

$$\frac{\alpha_1 \wedge \ldots\ldots. \wedge \alpha_i \wedge \ldots\ldots.. \wedge \alpha_m}{\alpha_i}$$

Từ một hội ta đưa ra một nhân tử bất kỳ của hội .

5. Luật đưa vào hội

$$\frac{\alpha_1, \ldots\ldots., \alpha_i, \ldots\ldots.. \alpha_m}{\alpha_1 \wedge \ldots\ldots. \wedge \alpha_i \wedge \ldots\ldots. \wedge \alpha_m}$$

Từ một danh sách các công thức, ta suy ra hội của chúng.

6. Luật đưa vào tuyển

$$\frac{\alpha_i}{\alpha_1 \vee \ldots\ldots. \vee \alpha_i . \vee \ldots\ldots. \vee \alpha_m}$$

Từ một công thức, ta suy ra một tuyển mà một trong các hạng tử của các tuyển là công thức đó.

7. Luật giải

$$\frac{\alpha \vee \beta, l\beta \vee \gamma}{\alpha \vee \gamma}$$

Từ hai tuyển, một tuyển chứa một hạng tử đối lập với một hạng tử trong tuyển kia, ta suy ra tuyển của các hạng tử còn lại trong cả hai tuyển.

Một luật suy diễn được xem là *tin cậy* (secured) nếu bất kỳ một mô hình nào của giả thiết của luật cũng là mô hình kết luận của luật. Chúng ta chỉ quan tâm đến các luật suy diễn tin cậy.

Bằng phương pháp bảng chân lý, ta có thể kiểm chứng được các luật suy diễn nêu trên đều là tin cậy. Bảng chân lý của luật giải được cho trong hình 5.3. Từ bảng này ta thấy rằng , trong bất kỳ một minh họa nào mà cả hai giả thiết $\alpha \vee \beta$, $l\beta \vee \gamma$ đúng thì kết luận $\alpha \vee \gamma$ cũng đúng. Do đó luật giải là luật suy điễn tin cậy.

α	β	γ	$\alpha \vee \beta$	$l\beta \vee \gamma$	$\alpha \vee \gamma$
False	False	False	False	True	False
False	False	True	False	True	True

False	True	False	True	False	False
False	True	True	True	True	True
True	False	False	True	True	True
True	False	True	True	True	True
True	True	False	True	False	True
True	True	True	True	True	True

Hình 5.3 Bảng chân lý chứng minh tính tin cậy của luật giải.

Ta có nhận xét rằng, luật giải là một luật suy diễn tổng quát, nó bao gồm luật Modus Ponens, luật Modus Tollens, luật bắc cầu như các trường hợp riêng. (Bạn đọc dễ dàng chứng minh được điều đó).

Tiên đề định lý chứng minh.

Giả sử chúng ta có một tập nào đó các công thức. Các luật suy diễn cho phép ta từ các công thức đã có suy ra công thức mới bằng một dãy áp dụng các

luật suy diễn. Các công thức đã cho được gọi là các *tiên đề* . Các công thức được suy ra được gọi là các *định lý*. Dãy các luật được áp dụng để dẫn tới định lý được gọi là một *chứng minh* của định lý. Nếu các luật suy diễn là tin cậy, thì các định lý là hệ quả logic của các tiên đề.

Ví dụ: Giả sử ta có các công thức sau :

Q ∧ S => G v H (1)

P => Q (2)

R => S (3)

P (4)

R (5)

Từ công thức (2) và (4), ta suy ra Q (Luật Modus Ponens) . Lại áp dụng luật Modus Ponens, từ (3) và (5) ta suy ra S . Từ Q, S ta suy ra Q∧S (luật đưa vào hội). Từ (1) và Q∧S ta suy ra G v H. Công thức G v H đã được chứng minh.

Trong các hệ tri thức, chẳng hạn các hệ chuyên gia, hệ lập trình logic,..., sử dụng các luật suy diễn người ta thiết kế lên các *thủ tục suy diễn_* (còn được gọi là *thủ tục chứng minh*) để từ các tri thức trong cơ sở tri thức ta suy ra các tri thức mới đáp ứng nhu cầu của người sử dụng.

Một *hệ hình thức* (formal system) bao gồm một tập các tiên đề và một tập các luật suy diễn nào đó (trong ngôn ngữ biểu diễn tri thức nào đó).

Một tập luật suy diễn được xem là *đầy đủ*, nếu mọi hệ quả logic của một tập các tiên đề đều chứng minh được bằng cách chỉ sử dụng các luật của tập đó.

Phương pháp chứng minh bác bỏ

Phương pháp chứng minh bác bỏ (refutation proof hoặc proof by contradiction) là một phương pháp thường xuyên được sử dụng trong các chứng minh toán học. Tư tưởng của phương pháp này là như sau : Để chứng minh P đúng, ta giả sử P sai (thêm ⌉P vào các giả thiết) và dẫn tới một mâu thuẫn. Sau đây ta sẽ trình bầy cơ sở này.

Giả sử chúng ta có một tập hợp các công thức $G = \{G_1,.....,G_m\}$ ta cần chứng minh công thức H là hệ quả logic của G **.** Điều đó tương đương với chứng minh công thức $G_1 \wedge .... \wedge G_m$ -> H là vững chắc. Thay cho chứng minh $G_1 \wedge ..... \wedge G_m$ =>H là vững chắc, ta chứng minh $G_1 \wedge .... \wedge G_m \wedge$⌉H là không thỏa mãn được. Tức là ta chứng minh tập G'‘= ($G_1,.......,G_m,$⌉H) là không thỏa được nếu từ G‘ta suy ra hai mệnh đề đối lập nhau. Việc chứng minh công thức H là hệ quả logic của tập các tiêu đề G bằng cách chứng minh tính không thỏa được của tập các tiêu đề được thêm vào phủ định của công thức cần chứng minh, được gọi là chứng minh bác bỏ.

5.5 Luật giải, chứng minh bác bỏ bằng luật giải

Để thuận tiện cho việc sử dụng luật giải, chúng ta sẽ cụ thể hoá luật giải trên các dạng câu đặc biệt quan trọng.

*Luật giải trên các câu tuyển

A1 v. . ... vAm v C

⌉C v B1 v.. .... v Bn

A1 v.. .... v Am v B1 v.... v Bn

trong đó Ai, Bj và C là các literal.

*   Luật giải trên các câu Horn:

Giả sử Pi, Rj, Q và S là các literal. Khi đó ta có các luật sau :

P1 ^. ....^Pm ^ S => Q,

R1 ^. ....^ Rn => S

P1 ^........^Pm ^ R1 ^...... ^ Rn =>Q

Một trường hợp riêng hay được sử dụng của luật trên là :

P1 ^..............^ Pm ^ S => Q,

S

P1 ^................^Pm   => Q

Khi ta có thể áp dụng luật giải cho hai câu, thì hai câu này được gọi là *hai câu giải được* và kết quả nhận được khi áp dụng luật giải cho hai câu đó được gọi là *giải thức* của chúng. Giải thức của hai câu A và B được kí hiệu là res(A,B). Chẳng hạn, hai câu tuyển giải được nếu một câu chứa một literal đối lập với một

literal trong câu kia. Giải thức của hai literal đối lập nhau (P và ˥P) là câu rỗng, chúng ta sẽ ký hiệu câu rỗng là [] , câu rỗng không thoả được.

Giả sử G là một tập các câu tuyển (Bằng cách chuẩn hoá ta có thể đưa một tập các công thức về một tập các câu tuyển). Ta sẽ ký hiệu R(G) là tập câu bao gồm các câu thuộc G và tất cả các câu nhận được từ G bằng một dãy áp dụng luật giải.

Luật giải là luật đầy đủ để chứng minh một tập câu là không thỏa được. Điều này được suy từ định lý sau :

Định lý giải:

Một tập câu tuyển là không thỏa được nếu và chỉ nếu câu rỗng [] $\in$ R(G).

Định lý giải có nghĩa rằng, nếu từ các câu thuộc G , bằng cách áp dụng luật giải ta dẫn tới câu rỗng thì G là không thỏa được, còn nếu không thể sinh ra câu rỗng bằng luật giải thì G thỏa được. Lưu ý rằng, việc dẫn tới câu rỗng có nghĩa là ta đã dẫn tới hai literal đối lập nhau P và ˥P (tức là dẫn tới mâu thuẫn).

Từ định lý giải, ta đưa ra thủ tục sau đây để xác định một tập câu tuyển G là thỏa được hay không . Thủ tục này được gọi là thủ tục giải.

procedure Resolution ;

Input : tập G các câu tuyển ;

begin

1.*Repeat*

1.1 Chọn hai câu A và B thuộc G ;

1.2 *if* A và B giải được *then* tính Res (A,B) ;

1.3 *if* Res (A,B) là câu mới *then* thêm Res (A,B) vào G ;

until

nhận được [] hoặc không có câu mới xuất hiện ;

2. *if* nhận được câu rỗng *then* thông báo G không thoả được

e lse thông báo G thoả được ;

end;

Chúng ta có nhận xét rằng, nếu G là tập hữu hạn các câu thì các literal có mặt trong các câu của G là hữu hạn. Do đó số các câu tuyển thành lập được từ các literal đó là hữu hạn. Vì vậy chỉ có một số hữu hạn câu được sinh ra bằng luật giải. Thủ tục giải sẽ dừng lại sau một số hữu hạn bước.

Chỉ sử dụng luật giải ta không thể suy ra mọi công thức là hệ quả logic của một tập công thức đã cho. Tuy nhiên, sử dụng luật giải ta có thể chứng minh được một công thức bất kì có là hệ quả của một tập công thức đã cho hay không

bằng phương pháp chứng minh bác bỏ. Vì vậy luật giải được xem là *luật đầy đủ cho bác bỏ.*

Sau đây là thủ tục chứng minh bác bỏ bằng luật giải

Procedure Refutation_Proof ;

input : Tập G các công thức ;

Công thức cần chứng minh H;

Begin

1. Thêm ⌉H vào G ;

2. Chuyển các công thức trong G về dạng chuẩn hội ;

3. Từ các dạng chuẩn hội ở bước hai, thành lập tạp các câu tuyển g' ;

4. áp dụng thủ tục giải cho tập câu G' ;

5. *if* G' không thoả được *then* thông báo H là hệ quả logic

   else thông báo H không là hệ quả logic của G ;

end;

Ví dụ: Giả giử G là tập hợp các câu tuyển sau

⌉A v ⌉B v P (1)

$\neg C \vee \neg D \vee P$ (2)

$\neg E \vee C$ (3)

A (4)

E (5)

D (6)

Giả sử ta cần chứng minh P. Thêm vào G câu sau:

$\neg P$ (7)

áp dụng luật giải cho câu (2) và (7) ta được câu:

$\neg C \vee \neg D$ (8)

Từ câu (6) và (8) ta nhận được câu:

$\neg C$ (9)

Từ câu (3) và (9) ta nhận được câu:

$\neg E$ (10)

Tới đây đã xuất hiện mâu thuẫn, vì câu (5) và (10) đối lập nhau. Từ câu (5) và (10) ta nhận được câu rỗng []. Vậy P là hệ quả logic của các câu (1) --(6).